Runtime Support for Task Migration on Distributed Memory Architectures

Yvon JÉGOU
IRISA / INRIA
Campus de Beaulieu
35042 RENNES Cedex, FRANCE
e-mail: Yvon.Jegou@irisa.fr
phone: (+33) 2 99 84 73 27
fax: (+33) 2 99 84 71 71

Abstract

The use of the task migration paradigm has been shown to allow efficient execution of unstructured codes on distributed-memory, parallel architectures. With this model, the data distributed on the parallel processors are never moved. When access to a non-local variable is necessary, the current computation is suspended and then resumed on the processor in charge of this variable. Our implementation of this model is based on a program transformation system and on a runtime library in charge of the communications. The program transformation part is architecture independent and considers a buffer space where the data of the migrating computations are stored. The associated runtime library is in charge of moving data among the processors, managing the buffer space inside a limited memory, and detecting the end of the computation. The implementation of this library depends heavily on the target architectures.

1 Introduction

The task migration paradigm manages the parallelism of programs at a fine grain level, in general a few instructions [5, 9]. A parallel language construct is transformed into the execution of small independent tasks. For instance, each task corresponds to one iteration of a parallel loop. In our implementation, the tasks migrate on the processors in order to read or write the data disseminated on the parallel memories. On distributed memory architectures, the migrations require the communication of messages between the processors. A good overlap of computation and communication is expected as soon as enough parallel tasks have been created.

In order to obtain efficient executions, complex interactions between the program and the communication layer must be avoided for each task migration. In our implementation, the data of tasks are appended to large buffers made available to the program by the runtime library. The application program and the runtime system interact only at the communication buffer level.

The typical targets of our system are numerical applications coded in standard programming languages with object distribution specifications such as Fortran HPF [8]. Our compilation technique for these programs replaces each code section containing accesses to distributed variables — parallel loops, in general — by a migration manager containing a task creation part and a task handler. The task creation part is in charge of initiating the execution of the tasks until they migrate, or until they end. The task handler part is in charge of executing the tasks received after a migration.

At runtime, the application code and the non-distributed variables are duplicated on the processors. A single execution thread is associated to each processor. The execution of the program alternates between duplicated phases when executing outside the migration managers and migrating phases when executing inside the migrating managers. During a duplicated phase, all the processors execute the same instructions and store the same values in the same variables. The programs never communicate or synchronize during a duplicated phase. During a migrating phase, each processor alternates between the execution of the task initiation part of the migration manager and the handler part upon the reception of messages containing tasks.

This execution scheme implements an asynchronous producer-consumer model of communication between the parallel processes. The number of tasks or messages exchanged is not known before the execution. Moreover, because of efficiency considerations, the memory space used for the communications must be strictly limited to the physical memory available at run time. The runtime system for task migration is in charge of the strict control of the flow of messages and of detecting the termination of each migration manager.

The task migration paradigm is introduced in Section 2. Section 3 describes the interface between the application and the runtime library in charge of implementing the machine-dependent part of the system. The typical targets of our system are numerical applications coded in standard programming languages with object distribution specifications such as Fortran HPF [8]. Section 4 shows the structure of the application code after transformation of parallel loops. The duties of the runtime system are described in Section 5 and some experimental results are shown in Section 6. Section 7 compares the task migration technique to other implementations where the idea of migrating the computation have been used and Section 8 concludes this paper.

2 The task migration paradigm

Each task generated is in charge of executing a chunk of the program. The tasks present in the system never synchronize: they are independent. A task can fork independent subtasks, but cannot communicate or synchronize with these subtasks. During its execution, a task can only access data from local memory. If the task needs to read or write variables located in the memory of some other processor, it must migrate to that processor. Writing to a distant memory location can be permitted when implemented by the target architecture. Such a case corresponds to forking an independent subtask in charge of storing some value in a variable. Each migration implies the transfer of all private data of the task to the destination processor in order to resume its execution.

In general, tasks are generated by the execution of parallel loops of high-level languages: each task is in charge of the execution of one iteration of the loop. The execution of a program based on the task migration paradigm alternates between duplicated phases where all the processors execute the same instructions on the same data (in general, scalar variables) and migrating phases where the processors share the execution of a set of independent tasks. Distributed variables can only be accessed during the migrating phases.

Alternating between duplicated phases and migrating phases allows a very light-weight implementation of the tasks: only the local data of the task need to move during migration. The control variables of the program are updated during the duplicating phases, their values never migrate. Since tasks never synchronize, the execution of a task is never suspended until the task ends or migrates. The current state of a task is completely defined by its private data. The creation of a new task — initiating the execution of a new iteration of a parallel loop — does not introduce any extra cost when compared to the classical sequential execution. No memory space, no execution stack, and no global identification needs to be associated with the task. The private data of the task are stored in the local space of the current thread. As long as the accesses to distributed variables are local to the processor, the task executes the same instruction sequence as in the sequential case. The only difference is that a locality test has been inserted by the compiler before each access to a distributed variable.

On the failure branch of a locality test, the code generated for the migration terminates the current iteration. The execution of the next iteration is initiated. A similar scheme is used for the execution of tasks received by the processor. Each iteration of the task handler activates one task from the received message. The task is executed until it ends or until it migrates. The private data computed during the execution of a task handler are stored in the local space of the execution thread. No extra memory space is allocated for this execution.

3 Runtime support for the task migration paradigm

The runtime library for task migration is in charge of the communications induced by task migrations. Because the tasks of our system are very light-weight, and often correspond to just a few instructions of a high level language, the number of instructions executed for each migration must be minimized in order to obtain efficient executions. The application and the runtime system communicate mainly through common variables. When a task migrates, its local data are appended to the buffer associated to the destination processor. The runtime system need to be called only if the space left in this buffer is not large enough. With large buffers, thousands of tasks can migrate without calling a runtime routine. Figure 1 shows the most important parts of the runtime interface. The application and the runtime library communicate mainly through the following common variables and functions:

vector `TabMes`**:**
: element `p` of `TabMes` identifies the buffer where the tasks destined to processor `p` are appended. This vector is modified only during calls to the runtime library.

flag `NotTerminated`**:**
: reset to false by the termination algorithm when the end of execution is detected. The termination algorithm is activated during calls to the runtime library.

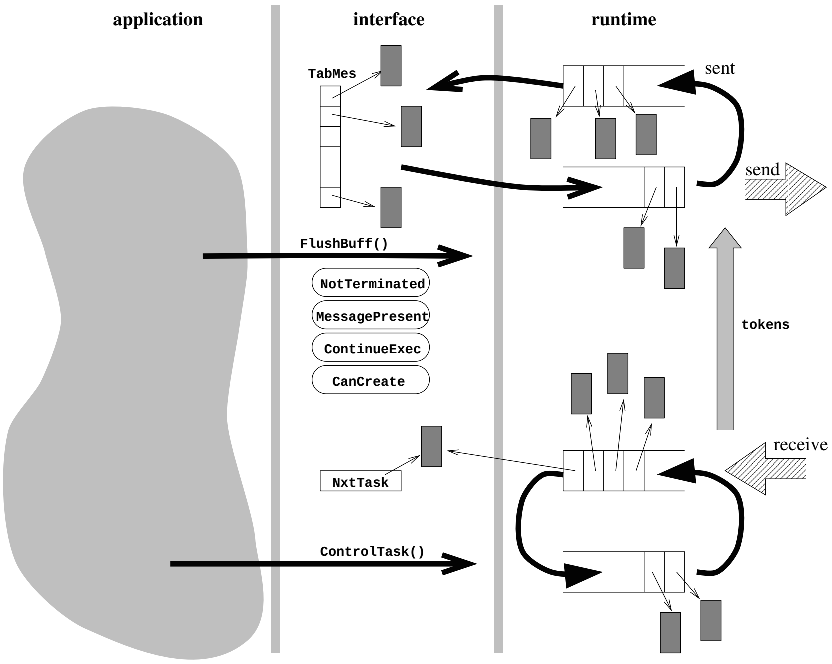

Figure 1: runtime interface

flag `MessagePresent`**:**
: when true, a buffer containing tasks is present and the first task waiting for execution is located at index `NxtTask` in the buffer memory. This flag is updated during calls to the function `RunTime()`.

flag `ContinueExec`**:**
: when true, the execution of the task located at `NxtTask` can be resumed. This flag is set during the execution of the function `RunTime()` when an input buffer is present and if the flow control permits migrations. It is reset by the last task of the buffer (call to the function `ControlTask()`) or by the flow control during a call to `FlushBuf(p)` (Section 5.1).

flag `CanCreate`**:**
: when true, allows the creation of new tasks. Managed by the flow control policy.

index `NxtTask`**:**
: when `MessagePresent`, location in `Mem` of the next migrated task.

counter `C`**:**
: the execution of the original loop is suspended every `Period` iterations in order to call the function `RunTime()`. This variable is managed by the application, but can be set to 0 during a call to `FlushBuf(p)` when the creation of new tasks must be stopped (flow control).

function `NewMig()`**:**
: called by the application when entering a new parallel construct.

function `RunTime()`**:**
: activates the runtime system, checks for the communication requests and activates the termination detection. This function must be called periodically.

function `ControlTask()`**:**
: called when the next task waiting for execution is a system task — the last task of the buffer, a termination task, a debug task — which is executed inside the runtime library.

function `InitiateTermination()`**:**
: the processor has initiated the execution of all the tasks it was allocated: all these tasks are terminated or have migrated. But messages containing tasks can still be received. The termination detection algorithm is started inside the runtime library. This distributed algorithm uses the relations between the messages in order to detect when the last message for the current migration manager is received.

function `BackGround()`**:**
: wait until the state of the runtime system changes.

function `FlushBuf(p)`**:**
: the buffer associated with processor `p` is transmitted to the communication layer and a free buffer is allocated

```
call NewMig()                      →  initialize variables in the runtime system;
i= B                               →  initialize loop counter;
do while(NotTerminated)            →  this external loop is executed until termination is detected;
 call RunTime()                    →  check for messages, handle termination;
 if(MessagePresent) then           →  a message has been received; the next task is at location
                                      NxtTask in buffer space Mem;
                                   ⇒  task handler
  do while(ContinueExec)           →  iterate on the tasks of the current message;
   goto(10,12,18,...)Mem(NxtTask)  →  select the code section corresponding to the task;
   call ControlTask()              →  it is a control task (debug, statistics, system, termination)
   goto 4                          →  end of this task;
10 continue                        →  code for task labeled 10;
  ....
   NxtTask= NxtTask+1              →  end of task labeled 10;
   goto 4
12 continue                        →  code for task labeled 12;
  ....
4  continue                        →  iterate on the current buffer;
  end do
 else if(CanCreate.and.(i.lt.N))   →  create new tasks, if possible;
                          then     ⇒  task creation part
  C= Period                        →  returns to the main loop after period iterations;
  do while((i.lt.N).and.(C.gt.0))
   ...                             →  body of initial loop;
   i= i+1
   C= C-1
  end do
  if(i.gt.N) then                  →  the last iteration has been executed. Initiate the termination al-
   call InitiateTermination()         gorithm;
  end if
 else
  call BackGround()                →  background task, idle loop;
 end if
end do
```

Figure 2: migration manager

in `TabMes(p)`. A call to this function can reset flags `CanCreate` and `ContinueExec`, and reset counter `C` if no more buffer space is available (flow control).

4 Code transformation and task migration implementation

Each parallel loop of the program is replaced by a while loop, the migration manager — see Figure 2 — containing a task handler and a task creation part. Each processor executes this loop until the logical variable `NotTerminated` becomes `.FALSE.` Each processor is in charge of initiating a subset of the iterations of the parallel loop. In general, the distribution of the iterations on the processors can be aligned on the distribution of the first array accessed inside the loop body in order to reduce the number of migrations. In our example, a processor initiates the iterations between the local values `B` and `N`.

4.1 Code for task migration

From the code generator point of view, a task migration is implemented by a sequence of instructions storing the current task context into some memory space associated with the destination processor. In the Fortran version of the runtime library, the communication buffers are allocated inside a large vector, `Mem`, declared by a Fortran common. The buffer descriptors and the task values are located in this vector. For instance, the location of buffer `b` where the next task will be stored is `Mem(Next(b))`. This technique avoids the introduction of non standard pointers in Fortran 77.

The code section in Figure 3 is generated for a task migration. The execution of this section does not involve any call to the runtime library, except when the space left in the buffer is not large enough for the whole task context.

`b= TabMes(i)`	→	get buffer descriptor for processor `i` from array `TabMes`;

`if((Next(b)+Sz).LT.Last(b)) then`
`  call FlushBuf(i)`
`  b= TabMes(i)`
`end if`
	→	check if enough space is available in the current buffer for `i`. If not, the call to `FlushBuf(i)` transfers the buffer to the communication layer and allocates a new buffer;

`p= Next(b)` `Next(b)= p+Sz`	→	the task context will be stored after location `p` in `Mem`. The next task will be appended at location `p+1`;
`Mem(p)= task state`	→	current state of the task;

`Mem(p+1)= ...`
`Mem(p+2)= ...`
`...`
`Mem(p+Sz-1)= ...`
	→	task context

Figure 3: code for a migration

The buffer descriptors `Next` and `Last` are local to each processor and shared between the application and the runtime library. The management of the buffer space, `Mem`, depends on the target architecture. On some NUMA architectures, it is possible to map different virtual pages of vector `Mem` on the local memories of different processors. In this case, a task migration is implemented using distant memory writes. However, each buffer is made available to the destination processor only after a call to the `FlushBuf` routine. This technique will be experimented in the near future. When messages are the only mean of communication between the processors, each task migration is split into two phases. During the first phase, the task context is stored into the buffer associated to the destination processor. This buffer is located in the memory of the sender. A call to the `FlushBuf` routine transmits the buffer to the communication layer of the runtime library, which is in charge of sending the messages. A large number of task contexts can be transfered inside the same message.

4.2 Example

Consider the following HPF program fragment:

```
      integer p(nt)
      real a(ns), b(nt)
!HPF$ Template vect1(nt)
!HPF$ Template vect2(ns)
!HPF$ Align (:) with vect1(:) :: b, p
!HPF$ Align (:) with vect2(:) :: a
!HPF$ Distribute (Block) onto Processors ::  vect1, vect2
      ...
      do i= 1, n
         a(p(i))= a(p(i))+b(i)
      enddo
```

In this program, vectors `a`, `b` and `p` are block-distributed on the processors. The mapping of each block-distributed template is described on each processor by three integer values: the bounds of the local block and the block size. For instance the mapping of template `vect1` is described by values `BlockSz1`, `BlockBegin1` and `BlockEnd1`. The value `BlockSz1` is the size of each block and is the same on all the processors. This value is used for the identification of the processor in charge of an element. In our example, element `b(i)` is owned by processor `(i-1)/BlockSz1`. `BlockBegin1` and `BlockEnd1` are the index bounds of the block owned by the current processor. Element `b(i)` is local to the current processor if `BlockBegin1`≤`i`≤`BlockEnd1`. A locality test for `b(i)` can be coded by the Fortran conditional instruction

```
       if((i.LT.BlockBegin1).OR.(i.GT.BlockEnd1)) goto 100
```

When it is local, element `b(i)` of the original code is accessed with `b(i-BlockBegin1+1)`. Vector `a` is aligned on template `vect2` described by `BlockSz2`, `BlockBegin2` and `BlockEnd2`.

Following the transformations patterns of Figures 2 and 3, the code of Figure 4 is generated for our loop. In this transformation, the iteration space of the loop has been aligned according to the template associated with the first distributed access inside the loop body: each processor is in charge of initiating the loop iterations such that element `p(i)` is local (`BlockBegin1`≤`i`≤`BlockEnd1`). This optimization avoids a locality test before acessing `p(i)`. Such automatic optimizations are frequently possible. In this code, `IMem` and `FMem` are mapped on the same address space `Mem` (Fortran `EQUIVALENCE` instruction), and allow mixing integer and floating-point values in the same messages. Only three memory words need to be transmitted for each migration.

```
      call NewMig()
      i= max(BlockBegin1, 1)
      do while(NotTerminated)
         call RunTime()
         if(MessagePresent) then
            do while(ContinueExec)
               goto(1) IMem(NxtTask)
               call ControlTask()
               goto 4
1              continue
               a(IMem(NxtTask+1)-BlockBegin2+1)=
                  a(IMem(NxtTask+1)-BlockBegin2+1)+b(FMem(NxtTask+2))
               NxtTask= NxtTask+3
               goto 4
4              continue
            end do
         else if(CanCreate.and.(i.le.min(BlockEnd1,n))) then
            C= Period
            do while((i.le.min(BlockEnd1,n)).and.(C.gt.0))
               if((p(i-BlockBegin1+1).LT.BlockBegin2)
               .OR.(p(i-BlockBegin1+1).GT.BlockEnd2)) goto 100
               a(p(i-BlockBegin1+1)-BlockBegin2+1)=
                  a(p(i-BlockBegin1+1)-BlockBegin2+1)+b(i-BlockBegin1+1)
               goto 101
100            continue
               dest= (p(i-BlockBegin1+1)-1)/BlockSz2
               buf= TabMes(dest)
               if((Next(buf)+3).LT.Last(buf)) then
                  call FlushBuf(dest)
                  buf= TabMes(dest)
               end if
               pt= Next(buf)
               Next(buf)= pt+3
               IMem(pt)= 1
               IMem(pt+1)= p(i-BlockBegin1+1)
               FMem(pt+2)= b(i-BlockBegin1+1)
101            continue
               i= i+1
               C= C-1
            end do
            if(i.gt.min(BlockEnd1,n)) then
               call InitiateTermination()
            end if
         else
            call BackGround()
         end if
      end do
```

Figure 4: generated code

5 Duties of the runtime system

The task migration runtime is in charge of the transmission of the output buffers to the destination processors, the management of the buffer space, avoiding deadlocks, and detecting the termination of the computation.

5.1 Flow control

In order to avoid costly memory management (page swapping), the whole parallel application must fit inside the physical memories of the parallel architecture. Before sending a large message, it is necessary to check that enough space is available for this message on the destination processor. The communication layer of the task migration runtime implements strict flow control through the use of tokens. Each token represents one input buffer in the destination processor. In general, the tokens and the requests for tokens are piggybacked on the messages. The tokens and the requests for tokens can also be transmitted with short system messages. System messages are generated when no task message can be sent because no token is available. System messages are also generated when there is no pending message for the same destination in order to avoid sending empty messages.

5.2 Deadlocks

All the computations of a processor are stopped as soon as no more free output buffers are available (interface flag `ContinueExec`). In general, this situation results from a lack of corresponding tokens. Once the computations are stopped on some processor, this processor does not free any more input buffers. No more tokens become available. This situation can result in an indefinite deadlock if many processors are stopped. Such deadlock situations are handled in the current implementation at the code generation level for the size of task contexts and by the runtime system through task creation control, processor priorities, and message misrouting.

Task size

When the size of the tasks increases during the computation, the limited size of the buffer space might not be sufficient to store the contexts of all existing tasks. In order to avoid this situation, the space that is reserved for a task inside an output buffer is the maximum size needed for the task in the future. So the space used at any time by the tasks under execution never increases. This "reservation" size is computed by the code generator.

Task creation

The runtime library implements the second step of deadlock avoidance through task creation control. When the number of output buffers waiting for transmission is higher than some predefined value, the creation of new tasks is stopped on the processor (the `CanCreate` flag is reset). Stopping task creation does not block the execution of existing tasks. In this state, the processor does not increase the space occupied by the tasks inside the buffer space.

Processor priority

The priority assigned by the runtime system to a processor is related to the ratio of the number of messages sent in the near past by this processor and the number of messages received. A processor consuming many messages but generating only few ones is assigned a high priority. This priority value is transmitted with all messages and is used for token distribution. When more tokens are requested than are available, the requests coming from higher priority processors are served first. If no token is available for a low priority processor, the number of messages waiting for transmission increases on this processor and blocks the creation of new tasks (task creation control). The combination of priority management and of task creation control allows global management of the entire buffer space.

Misrouting of messages

The last step in deadlock avoidance is applied by the communication layer of the runtime library. When all the output buffers are waiting for a token, a local deadlock situation is detected. One of the output buffers is selected and is sent to a processor for which a token is available. This last step frees some space on overloaded processors which can resume the execution of incomming tasks. A misrouted message is immediately transferred to the communication layer for transmission to the real destination.

When the migrations are uniformly distributed on the processors, the combination of task size reservation and task creation control ensures deadlock-free execution. But when only a few processors must handle all the generated tasks, it is possible that their buffer space is not large enough to store all existing tasks. Avoiding deadlocks even in such non-efficient code sections is important because the whole system is intended to be used through an automatic Fortran front end. With message misrouting, the last step of deadlock avoidance extends the available buffer space of loaded processors by the buffer space available on unloaded processors. Deadlock situations have never been met during our experimentations although the application of these four steps cannot be proved to ensure deadlock-free executions.

5.3 Termination

The runtime library is in charge of detecting the termination of a parallel construct. The precise description of this algorithm is outside the scope of this paper and can be found in [6]. Termination is computed during a variable

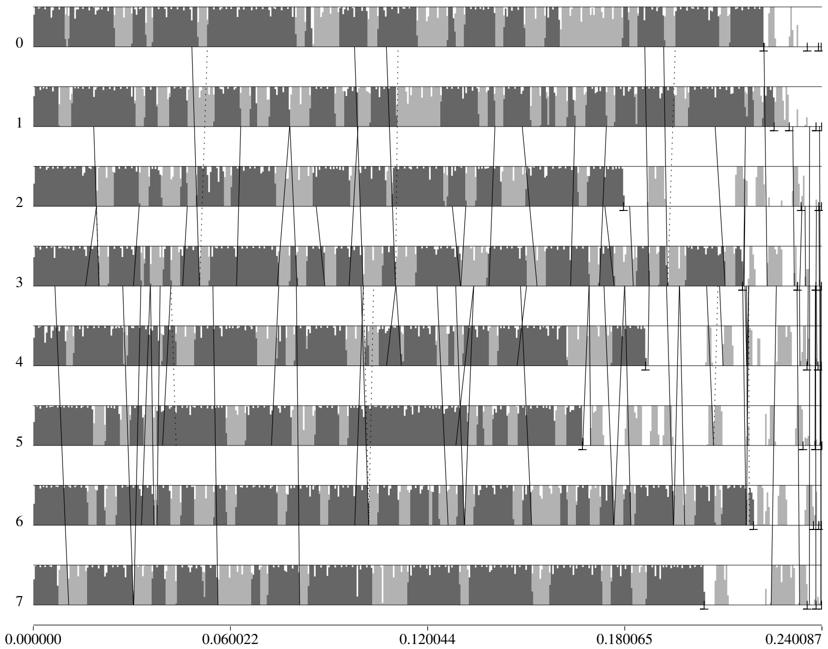

Figure 5: execution trace for 8 processors

number of phases once all the tasks have been created. During each phase, the algorithm tests if at least one task migrated during the preceding phase (tree reduction). At the end of each phase, each processor sends $log_2 N$ messages and flushes all non-empty buffers.

6 Experimentation on the Paragon X/PS

The task migration paradigm has been experimented with a numerical code for unstructured triangular meshes [2]. The migration managers of the program were automatically generated from the data-flow graphs of the parallel loops. A fully automatic Fortran front end is under development and will be available in the future. The resulting program was run on two versions of the same mesh. The original mesh shows poor memory-accesses locality: more than 90% of the iterations result in a migration. The optimized mesh was obtained by a classical, but simple renumbering algorithm; less than 8% of the iterations produce a migration.

The code was run on an Intel Paragon X/PS with 56 processors, running Mach/OSF1/1 Release 1.0.4 Server 1.3. The communication layer exchanges the messages through calls to the system `nx` communication library using non-blocking send and receive requests. The state of the waiting requests is checked during periodic calls to runtime routine `RunTime()`. The parallel execution efficiency on 32 processors is close to 50% for the optimized mesh and close to 20% for the original mesh. Figure 5 shows an execution trace for one iteration on eight processors with the original mesh. In this trace, the state of a processor is dark when executing inside the task creation part of the migration manager, grey when executing the task handlers, and white when executing inside the runtime library. The traces clearly show that the communications and the computation overlap. The waiting time appears only at the end of the computation and is mainly due to the unbalanced load. The lines show the messages exchanged by processor three. The dotted lines correspond to system messages. During the 0.24 seconds of this execution, processor three sent 1.2 Mbytes of data inside 41 messages. The average message length is 28 Kbytes and the maximum length is 50 Kbytes. This processor received 39 messages during this period. Three system messages (8 bytes) were also sent and six received during the run. The same execution on 32 processors is three times shorter (0.08 seconds) than in the eight processors case. Each processor sent around 329 Kbytes although the average number of messages produced by each processor is close to the eight processors case. These messages are destinated to 20 different processors in average. Such a large number is due to the poor locality of the accesses to distributed objects.

Figure 6 shows an execution trace on eight processors using the optimized mesh. This trace clearly shows two phases. During the first phase, there is no exchange between the processors. In fact, less than 8% of the tasks migrate and the output buffers are never filled. Once all the iterations have been initiated, the termination algorithm is started and flushes all the buffers containing tasks. During this iteration, each processor sends 50 Kbytes in

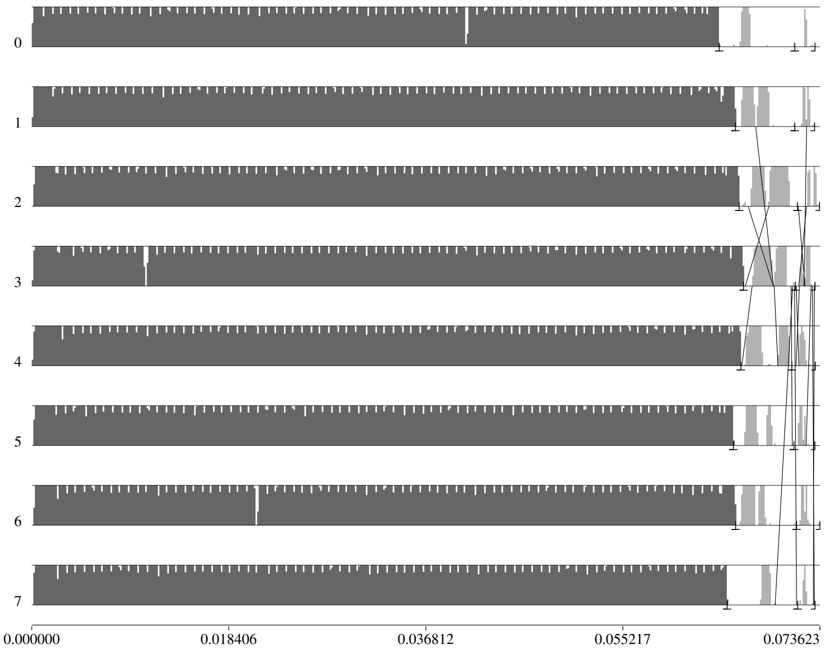

Figure 6: execution trace using the optimized mesh for 8 processors

average inside eight messages.

7 Related work

The idea of migrating the computation on the processors of a parallel architecture has been developped in various domains such as load distribution in distributed operating systems, implementation of (hybrid-)data-flow or implementation of object oriented applications on distributed architectures [7].

The problem of load distribution in a distributed operating system involves re-allocating the workload among the nodes of the system. Using process migration, already active processes can be moved from one machine to another [3]. A process migration requires copying the whole process address space on the destination processor. In our implementation, the tasks migrate in order to reach data located in the memories of the processors. The distribution of the tasks on the processors depends only on the distribution of the data. Process migration has also been proposed for maximizing the utilization of wokstations, for instance through checkpointing in [10].

Computation migration, [4], can be viewed as partial thread migration. This technique tries to increase the locality of memory accesses while reducing the amount of state that must be moved. Only a few activation frames at the top of the stack are moved. The tasks of our system are activated on top of the stack of the task handler. A migration does not move any activation frame, but just a few variable from the top of the stack.

The compilation and the execution of data-flow languages produce large numbers of independent threads. Classical implementations on distributed architectures are based on the use of remote procedures calls and continuations. Thread migration has been proposed in this context in order to exploit the spacial locality found in the usage of many data structures. For instance, the Nomadic Threads, [9], have been shown to use fewer data transfers and to take advantage of locality. The "execution unit" used in the Nomadic Threads is limited to a few high level language instructions, as it is also the case in our system. The main difference between the data-flow implementation and our is in the relations between the tasks. The data-flow implementation forms a hierarchy of tasks following the fork-join synchronization model through the use of continuations. Our implementation alternates duplicated phases of execution without any task and migrating phases with independant tasks. There is no relation between our tasks.

The implementation of object oriented applications on distributed architectures is another active domain where computation migration has been experimented as in [11].

In our implementation, a task is a "piece of work" represented inside the compiler by an extended data-flow graph. No stack or frame is associated to the task during execution. All the events occuring during the task execution have been predicted at compile-time. For instance, each migration is implemented by a sequence of instructions storing the current local data of the task inside some buffer. This technique allows a very light-weight implementation of small tasks. The main limitation of this technique is that all the migrations of the task must occur inside its body. Although subroutines can be called during execution, no migration is possible during such a call. This limitation restricts the application domain of our technique to the parallel execution of independant loop iterations. In fact, our technique is more related to the techniques used for the compilation of numerical programs for supercomputers (microtasking on Cray systems [1], for instance).

8 Conclusion

The task migration paradigm allows asynchronous implementation of parallel programs using a simple producer-consumer communication model. A clear definition of the relations between the application program and the runtime library avoids the introduction of architecture dependent — non portable — mechanisms in the application. All the architecture-dependent parts of this model — memory hierarchy, communication layer — are implemented in the runtime library. The experimental results show that the execution of irregular applications can be accelerated on distributed memory parallel architectures, even when they generate a large number of non-local accesses to the memories. The main targets of our system are massively parallel architectures with message passing communications. Our strategy will also be validated in the near future using different communication models, such as direct exchanges through shared memories, for the runtime system implementation.

References

[1] M. BOOTH, K. MISEGADES, *Microtasking: A New Way to Harness Multiprocessors*, Cray Channels 1986.

[2] M. O. BRISTEAU, J. ERHEL, P. FEAT, R. GLOWINSKY, J. PÉRIAUX, "Solving the helmotz equation at high wave numbers on a parallel computer with a shared virtual memory", *International journal of supercomputer applications and high performance computing*, (9.1), 1995.

[3] F. DOUGLIS, J. OUSTERHOUT, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software - Practice And Experience*, vol. 8(21), pp. 757–786, 1991.

[4] W. C. HSIEH, P. WANG, W. E. WEIHL, "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems", *in Proceedings Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 28(7), pp. 239–248, July 1993, New York, NY, USA.

[5] Y. JÉGOU, "The Migrating Tasks: An Execution Model for Irregular Codes", *in Proceedings of Euro-Par'96 Parallel Processing*, L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Editors), *Lecture Notes in Computer Science*, 1123, p. I:562–570, Lyon, France, august 1996.

[6] Y. JÉGOU, *Exécution de codes irréguliers par migration de tâches*, INRIA Research Report, No2436.

[7] M. R. ESKICIOGLU, L. F. CABRERA, *Process Migration, An Annotated Bibliography*, IBM Research Report RJ7935, 1991.

[8] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Technical Report revision 1.0, Rice University, May 1993.

[9] S. JENKS, J.-L. GAUDIOT, "Nomadic Threads: A Migrating Multithreaded Approach to Remote Memory Accesses in Multiprocessors", *in Proceedings of Parallel Architectures and Compilation Techniques, PACT'96*, Boston, pp. 2–11, October 1996.

[10] M. J. LITZKOW, M. LIVNY, M. W. MUTKA, "Condor – A Hunter of Idle Workstations", *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pp. 104–111, San Jose, Calif., June 1988.

[11] J. PLEVYAK, V. KARAMCHETI, X. ZHANG, A. A. CHIEN, "A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers", *in Proceedings Supercomputing '95*, San Diego, California.