Distributed Computations Driven by Resource Consumption

Luc Moreau

Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
L.Moreau@ecs.soton.ac.uk

Christian Queinnec

LIP 6 & INRIA-Rocquencourt
4, place Jussieu, 75252 Paris Cedex France
Christian.Queinnec@lip6.fr

Abstract

Millions of computers are now connected together by the Internet. At a fast pace, applications are taking advantage of these new capabilities, and are becoming parallel and distributed, e.g. applets on the WWW or agent technology. As we live in a world with finite resources, an important challenge is to be able to control computations in such an environment. For instance, a user might like to suspend a computation because another one seems to be more promising. In this paper, we present a paradigm that allows the programmer to monitor and control computations, whether parallel or distributed, by mastering their resource consumption. We describe an implementation on top of the thread library PPCR and the message-passing library Nexus.

1 Introduction

As we live in a world with finite resources, it is of paramount importance for the user to be able to monitor and control computations. This task is all the more complex since computations may be parallel, distributed, and most probably make use of code written by others. This problem is particularly illustrated by the Internet as the user wishes to search information over the WWW, exploits parallelism to improve efficiency, relies on distribution to increase locality, but also wants to concentrate the computing power in the most promising directions to reduce searching time.

There are two types of applications that we particularly wish to program. First, a user that has initiated a computation over the Internet should be able to suspend the computation in order to analyse the results he has already obtained. If these were unsatisfactory, s/he must be able to resume the computation from the point where it was suspended in order to collect the next results. In this example, computations should be understood as possibly parallel and distributed. Second, service providers offer computing facilities or resources to customers who subscribe to their service by transferring electronic cash [38]; in return, service providers supply them with a handle to submit jobs, create accounts that they debit according to the usage of the facilities, and inform users of the exhaustion of their account. Again, jobs submitted by customers may generate parallel and distributed computations which must be monitored by the service provider.

Our goal is to provide the means by which everybody, customers and service providers, can get the most anyone can out of a situation with bounded resources. We believe that parallel computations can be driven by mastering their resource consumption. In this paper, we present a new paradigm, called $\mathcal{Q}$uantum, that allows the user to monitor and control distributed computations. The basic principle of $\mathcal{Q}$uantum is to keep track of the resources consumed by computations. Resources can be understood as processors cycles, bandwidth and duration of communications, or even printer paper. We shall adopt a more generic view by saying that computations need *energy* to be performed[1]. Although the notion of energy is part of $\mathcal{Q}$uantum, the programmer cannot create energy ex nihilo, but can only transfer it between computations via some primitives. As a result, we were able to ensure a general principle for $\mathcal{Q}$uantum: given a finite amount of energy, any computation is finite.

$\mathcal{Q}$uantum generalises some approaches adopted in agent scripting languages to control resources [27, 40]. Besides its resource-oriented foundations, $\mathcal{Q}$uantum is designed for parallelism and distribution. Furthermore, $\mathcal{Q}$uantum was conceived to be independent of any programming language, and in particular, of the primitives for parallelism and distribution, and of the memory model (central, distributed, with or without coherence). In order to prove this claim, we have built a library that implements the primitives of $\mathcal{Q}$uantum on top of the thread library PPCR [52] and the message-passing library Nexus [13]. $\mathcal{Q}$uantum is also integrated with Nexeme [32], a distributed implementation of Scheme [47]. Internally, $\mathcal{Q}$uantum energy is modelled by counters that are decremented every time a thread is scheduled; asynchronous notifications inform the user of energy exhaustion and computation termination. Two other primitives are able to supply or remove energy.

This paper is organised of follows. We present the intuition of $\mathcal{Q}$uantum in Section 2. In Section 3, we focus on the implementation of $\mathcal{Q}$uantum. Finally, Section 4 discusses related work and is followed by a conclusion. A companion paper [33] describes the formal semantics of $\mathcal{Q}$uantum and present several examples using its primitives.

2 The Library $\mathcal{Q}$uantum: Rationale

In this section, we introduce $\mathcal{Q}$uantum, its primitives and their intuitive semantics, and the considerations that lead to its design. The abstract syntax of $\mathcal{Q}$uantum primitives is displayed in Figure 1.

$\mathcal{Q}$uantum is independent of the primitives for parallelism or distribution. Parallel threads of evaluation may be created using Posix threads [22], or higher-level constructs like **pcall** [34, 45] or **future** [18, 31]. In the sequel, we use the term *task* to denote an evaluation thread created by the constructs for parallelism. Similarly, message-passing style primitives [12, 14, 13] or higher-level forms of remote function calls [30, 32, 42] may be adopted for distribution.

[1] Other names found in the literature for a similar concept are fuel [20], computron [46, p. 102–103], teleclick [27], or metapill [1].

$$\begin{array}{rcl}
\mathit{primitives} & ::= & \textsf{call-with-group}(F, e, \varphi_e, \varphi_t) \\
 & & \mid \mathit{pause}(g, \varphi_p) \\
 & & \mid \mathit{awake}(g, e) \\
g & \in & \mathit{Group} \\
e & \in & \mathit{Energy} \\
F & : & \mathit{Group} \times \mathit{Energy} \rightarrow \alpha \\
\varphi_e, \varphi_t, \varphi_p & : & \mathit{Group} \times \mathit{Energy} \rightarrow \mathit{void}
\end{array}$$

Figure 1: Abstract Syntax of Quantum Primitives

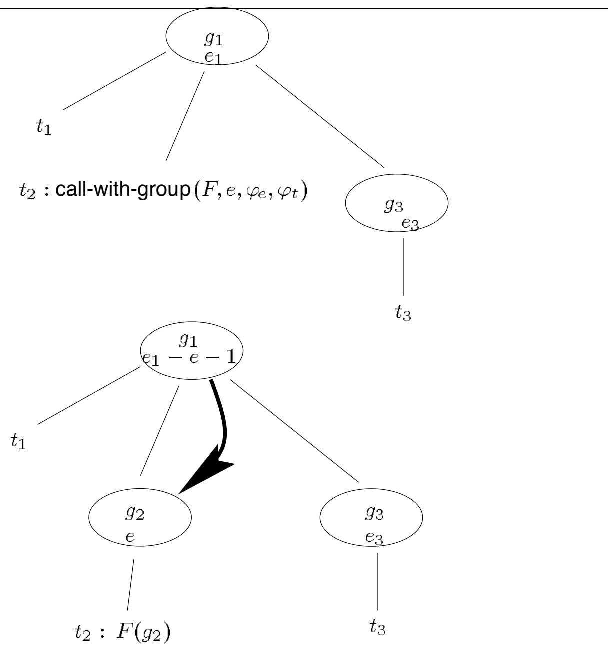


Figure 2: Group Creation

Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluations proceed. In our view, it is essential to be notified of the *termination* of a computation so that, for instance, unconsumed resources can be transferred to a more suitable computation. Similarly, we want to be informed of the *exhaustion* of the resources allocated to a computation, so that for instance more resources can be supplied.

In order to be notified of the termination or energy exhaustion of a computation, we need an entity that represents the computation. A *group* is an object that can be used to refer to a computation in a Quantum program. A group is associated with a computation composed of several tasks proceeding in parallel; in turn, they can initiate subcomputations by creating subgroups. As a result, our computation model is hierarchical. A group is said to *sponsor* [26, 37, 18] the computation it is associated with. Reciprocally, every computation has a sponsoring group, and so does every task.

At creation time, a group is given an *energy quota*. More specifically, a computation that evaluates the expression call-with-group$(F, e, \varphi_e, \varphi_t)$ under the sponsorship of a group g_1, creates a new first-class group g_2 that is allocated an initial quota of energy e and whose parent is g_1. Furthermore, it initiates a computation under the sponsorship of g_2 by calling F with g_2 and e as argument; hence, the user function F receives a handle on its sponsoring group. As Quantum keeps track of resource consumption, the energy e allocated to g_2 is deducted from the energy of g_1. Figure 2 displays the behaviour of the primitive call-with-group. We see a configuration where a group g_1 is sponsoring two tasks t_1 and t_2 and a subgroup g_3 itself sponsoring a task t_3. After evaluating the primitive call-with-group, a new subgroup g_2 sponsoring the application of F on g_2 and e is created; energy is transferred from g_1 to g_2. This transition assumes that $e_1 > e$.

g_1
e
t
g_1
$e - e_1$
t'

Figure 3: Computation and Energy Consumption

Remark In Quantum, every action has an associated cost. Figure 2 shows that the energy of g_1 is $e_1 - e - 1$ after transition. The value $-e$ is the amount of energy transferred to the new group g_2 and -1 represents the cost of the group creation operation. In order to simplify the presentation, we assume that all transitions have a *unitary* administrative cost. □

Quantum enforces the following principle: any computation consumes energy from its sponsoring group. Therefore, not only is a group perceived as a way of naming computations, but also it must be regarded as an *energy tank* for the computation. Figure 3 displays a task t evolving to state t' by performing an action, whose cost e_1 is charged to the sponsoring group g_1.

In addition, two events may be signalled during the lifetime of a group: *group termination* and *energy exhaustion* are asynchronously notified by applying the user functions (the *notifiers*) φ_t and φ_e, respectively[2]. A group is said to be terminated, when it has no subgroup and it does not sponsor any task; i.e. no more activity can be performed in the group. In Figure 4, when the only task t_4 of group g_3 is terminating, the function φ_t is asynchronously called with g_3 as argument to notify its termination, and the energy surplus of g_3 is transferred back to g_1. Note that the execution of the notifier φ_t is sponsored by g_1, i.e. the parent of g_3.

In Figure 5, a computation t_2 sponsored by g_2 requires more energy than available in g_2; the function φ_e is asynchronously called on g_2 to notify its energy exhaustion, also under the sponsorship of g_1, with transfer of the remaining energy of g_2 to g_1.

Remark An exhaustion notification, like every Quantum transition, has a cost. In order to guarantee that there is enough energy to notify any occurring exhaustion, we define the "exhaustion threshold" as the cost of notifying an exhaustion. An exhaustion notification will be raised if the cost of the current operation is higher than the remaining energy in its sponsoring group minus the cost of notification. □

Figure 6 displays the state transition diagram for groups. At creation time, a group is in the running state, which means that the tasks that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented

[2]Subscript t denotes termination, whereas subscript e denotes exhaustion.

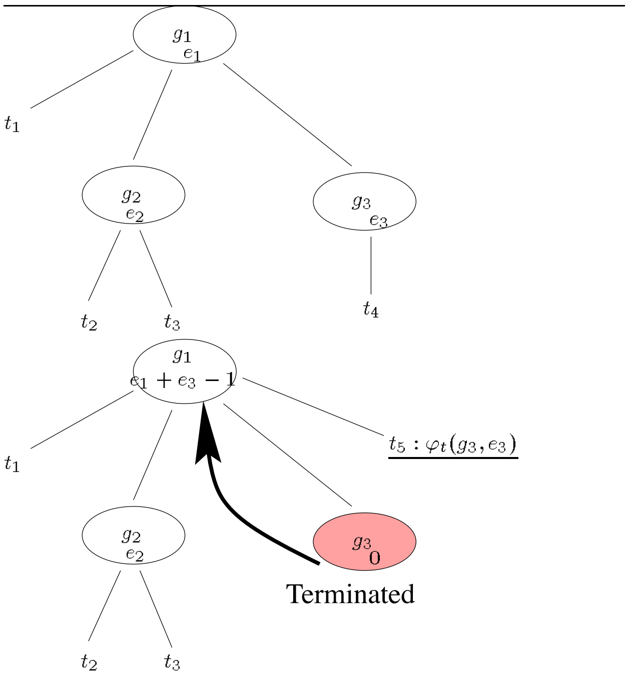


Figure 4: Termination of a Group

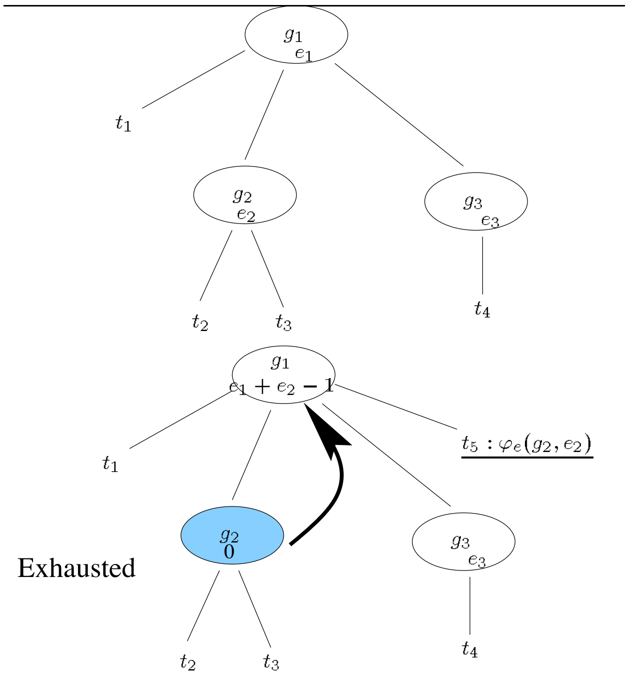


Figure 5: Exhaustion of a Group

by dotted lines. Once a computation requires more energy than available in its sponsoring group, the state of its group changes to exhausted, and at the same time an asynchronous notification φ_e is run. When all subgroups and all tasks sponsored by a group terminate, its state becomes terminated, while the asynchronous notifier φ_t is called. Let us observe that the terminated state is a dead end in the state diagram; this guarantees the *stability* of the termination property: once a computation terminates, it is not allowed to restart (as the resource that it did not consume may have been reallocated).

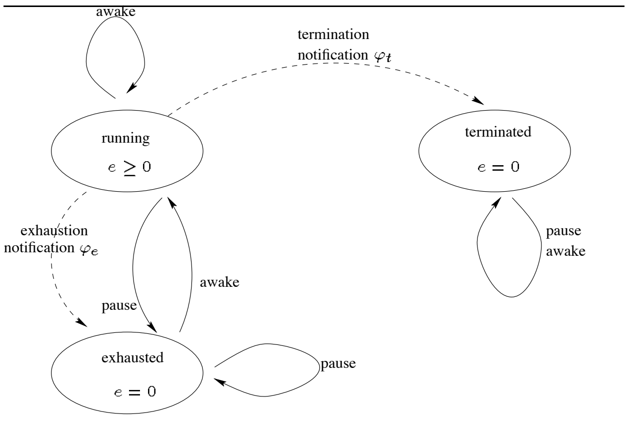


Figure 6: State Transitions

Energy may be caused to flow between groups, independently of the group hierarchy, under the control of the user program. Two primitives operate on groups: pause and awake. Intuitively, the primitive pause forces a running group *and its subgroups* into the exhausted state, and all the energy that was available in this hierarchy is transferred to the group that sponsored the pause action. The construct $awake(g, e)$ transfers energy e to the group g, after deducting it from the group sponsoring the awake action. If the group g is in the exhausted state, its state is changed to running; if the group is in the terminated state, awake acts as a null operation. Figure 7 displays the behaviour of awake, assuming that $e_1 > e$ and g_2 is not terminated.

Let us observe the non-symmetric behaviours of pause and

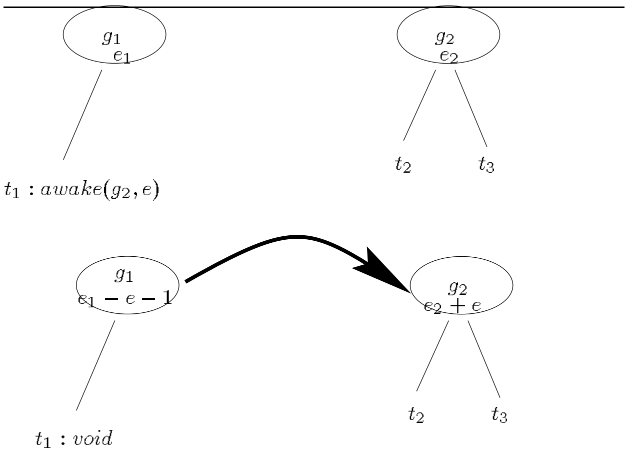


Figure 7: Awaking a Group

awake: the former operates recursively on a group hierarchy, while the latter acts on a group and not its descendants. However, we might wish to awake a hierarchy recursively, for instance when we wish to resume a paused parallel search. In particular, we might wish to resume the search with the energy distribution that existed when the hierarchy was paused. Unfortunately, such information is no longer available because groups are *memory-less*. By this, we mean that a group does not remember the amount of energy it had before being paused. It is therefore the programmer's responsibility to leave some information at pausing-time about the way a hierarchy should be awakened. Not only does pause transfer energy, but it also posts a notification for each group in the tree. Figure 8 displays the precise behaviour of pause. Evaluating $pause(g_1, \varphi_p)$ forces into the exhausted state each group g' in the hierarchy rooted by g_1; moreover, for each g', an evaluation that calls φ_p with g' as argument is created under the sponsorship of the parent of g'. Let us note that notifications are prevented to run as all groups in the hierarchy have been dried out (except the notification on the root g_1, which is sponsored by g_0, the parent of g_1 and then might run). Once the root of the hierarchy is awakened, any notification sponsored by the root will be activated, and may decide to awake the group it is applied on, and step by step, energy may be redistributed among the hierarchy.

Our permanent concern when designing $\mathcal{Q}$uantum was to be able to compute in a distributed framework. Hence, we decided that $\mathcal{Q}$uantum would be independent of the memory model: so, real shared memory, shared memory simulated over a distributed memory [30], distributed causally coherent memory [42] are memory models that may be adopted with $\mathcal{Q}$uantum.

3 Implementation Description

This section outlines the current implementation. First, it presents a solution in a non-distributed setting, and then describes specific problems encountered when modifying the thread library PPCR [52]. Second, it covers the distributed aspects, and then specific problems encountered when using the library for distribution Nexus [13].

3.1 Single Space Model

The single space parallel model is rather simple. Although tasks may not be first-class values in the adopted language (for instance, with the **future** approach [18]), they do exist within the implementation. They are created with the primitives for parallelism available in the current language or library, and as usual, a scheduler manages the set of all tasks. A task contains a reference to its sponsoring group.

Groups are organised hierarchically; a group knows *(i)* its parent group, *(ii)* its exhaustion and termination notifiers, *(iii)* the evolving set of *direct tasks* that is, the tasks that have this very group as sponsoring group, *(iv)* the evolving set of *direct* subgroups, that is, the groups that have this group as parent group.

Each group is also associated with a *tank of energy*. The energy spent by a running task is deducted from the tank of its sponsoring group. When a tank is exhausted, a notification is posted to the parent group. The scheduler prevents the tasks of a group with an empty tank to run.

When a group has no subgroup and looses its ultimate sponsored task, the group must be terminated: it is removed from the subgroups of its parent group and its tank is poured into the tank of its parent group.

Energy may flow independently of the hierarchy of groups using the pause and awake operations. In order to awake a target group with a given amount of energy, we remove this energy from the tank of the current group, and transfer it into the tank of the target group.

Pausing a group g is the most complex operation since it recursively dries the tanks of all the groups in the tree rooted at g. When pause empties the tank of a group g', it posts a notification in the parent group of g' mentioning the amount of stolen energy; the notifier is the function given as second argument to pause; the notification is a task whose job is to invoke the notifier with the paused group and the amount of stolen energy.

Transitions described in Figures 2, 3, 4, 5 and 7 are to be considered as atomic. On the other hand, the pause function is not atomic. The pause function returns after the complete visit of a hierarchy of groups. This visit is performed by one or more tasks that run concurrently with all other tasks. Note also that multiple notifications for a same group may be simultaneously running if a group was paused or awakened multiply. Notifiers are not run in mutual exclusion.

Let us note that the garbage collector should reclaim unreachable groups that are both exhausted and without subgroups. Indeed, such groups cannot be awakened since they are unreachable, and no task can be created under their sponsorship according to our model.

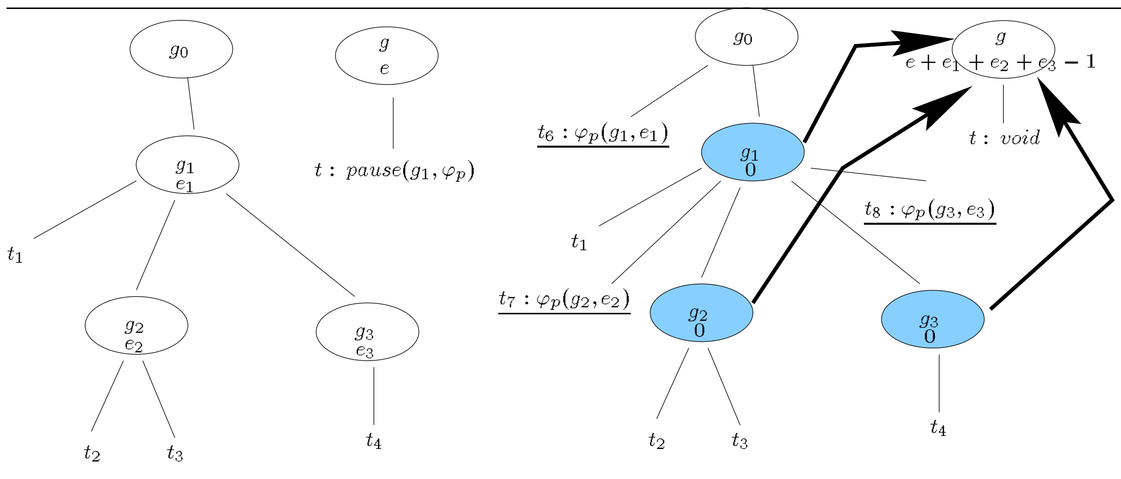
3.2 Threaded Implementation

We have implemented the single space model by modifying PPCR, the Portable Posix Common Runtime [52], a user-level preemptive thread package that comes with Boehm and Weiser's garbage collector [17]. We elected to modify this thread package as it was already adopted for NeXeme [32], a distributed dialect of Scheme based on the library Nexus [13].

A PPCR thread datastructure is given an extra field which points at its group. By default, the value of this field is NIL, which means that the current thread is not running under the sponsorship of a group, and that it has the usual PPCR semantics. If not NIL, the current thread is under sponsorship of a group. This approach allows us to write applications where some threads have the usual PPCR semantics, while others follow $\mathcal{Q}$uantum semantics. Every time a thread is scheduled, the energy of its associated group, if it exists, is decremented. Therefore, in the current implementation, a unit of energy is defined as a *time slice* of the PPCR scheduler.

The whole idea of $\mathcal{Q}$uantum is that the scheduler should preempt a thread every time its associated group has no energy left. However, like exceptions, preemption cannot be synchronous: if a thread is in a critical section, preempting it might lead to a deadlock. Therefore, we used a similar technique as for asynchronous signals in PPCR, where threads check for pending signals when exiting critical sections. Two extra-fields per thread are used for this purpose: a field indicating that $\mathcal{Q}$uantum preemptions are allowed, and another field indicating that a $\mathcal{Q}$uantum event is pending. When a thread exits a critical section, it checks if the $\mathcal{Q}$uantum event flag was set by the scheduler and if preemptions by $\mathcal{Q}$uantum are allowed.

According to the semantics, a new thread is created every time $\mathcal{Q}$uantum notifies an event. In order to reduce the cost of thread creation at notification time, we spawn a special thread whenever we create a new group; this special thread is associated with the group, and is nicknamed as the "group supervisor thread". When a user thread exits a critical section and detects a $\mathcal{Q}$uantum event, it awakens the group supervisor thread associated with its sponsoring group, which in turn runs the user specified specification. Care is taken that only one thread per group can awaken the group supervisor thread per $\mathcal{Q}$uantum event. A group supervisor thread is maintained alive as long as its associated group is still in its useful period. This organisation minimises changes in the PPCR kernel,



g_0
g
e
$t: pause(g_1, \varphi_p)$
g_1
e_1
t_1
g_2
e_2
g_3
e_3
t_2
t_3
t_4
g_0
g
$e + e_1 + e_2 + e_3 - 1$
$t_6 : \varphi_p(g_1, e_1)$
$t: void$
g_1
0
$t_8 : \varphi_p(g_3, e_3)$
t_1
$t_7 : \varphi_p(g_2, e_2)$
g_2
0
g_3
0
t_2
t_3
t_4

Figure 8: Pausing a Group

and $\mathcal{Q}$uantum preemption and resumption can be implemented as user-level code.

In addition, every primitive that accesses a critical section was modified so that $\mathcal{Q}$uantum preemption could not occur during their evaluation. In particular, this concerns I/O routines and the memory-allocation procedure `GC_malloc` [17].

This asynchronous notification slightly changes the semantics: indeed, energy may now become negative when the execution of a primitive overruns the energy left in its sponsoring group. We consider that it is not a real problem as the user is not given access to any construct that prevents $\mathcal{Q}$uantum preemptions; some primitives only, with a bounded length, are allowed to *delay* preemption.

The marginal cost of supporting $\mathcal{Q}$uantum is not high. Every time a thread is scheduled, the existence of a sponsoring group is tested, its associated energy is decremented, and a flag is set if it becomes negative.

3.3 Distributed Space Model

Distribution introduces multiple disjoint spaces of values linked by remote pointers and communicating by messages. We assume that distribution is introduced by a construct that creates a task on a remote site. Active message-passing style primitives [13] or remote function calls [30, 32, 42] are both compatible with $\mathcal{Q}$uantum. All sites have a scheduler managing local tasks.

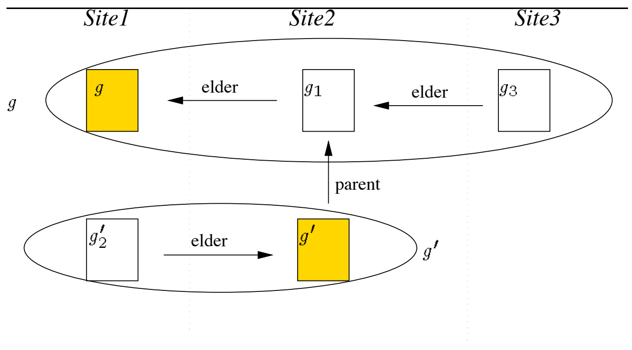


Figure 9: Local Groups and Brotherhood

The previously exposed implementation may be extended easily to cover distribution after introducing the notion of *group brotherhood*. A group stands for several tasks on several sites. To cope with distribution, a group is split into a hierarchical family of *local groups* ordered by a brotherhood relationship, each of them being local to a site. When a group g (see Figure 9) is created by call-with-group on *Site1*, it is known as the *eldest local group* of a family so far containing one brother only. When a task sponsored by g creates a remote task on *Site2*, a *younger local group* g_1 is created on *Site2* with g as elder brother. Symmetrically, g adds g_1 as younger brother. When a subgroup g' is created by call-with-group on *Site2* under the sponsorship of g_1, its parent will be g_1.

In summary, a *local group* belongs to a site and holds the following information: *(i)* its *elder* brother, *(ii)* its *direct younger* brothers, *(iii)* its direct local tasks, i.e. the evolving set of tasks that run under its sponsorship on the local site, *(iv)* its direct local subgroups, i.e. its locally created subgroups.

A group is represented by at most one local group per site; local groups are implementation entities, but only eldest local groups are first-class values that incarnate the group they stand for. Therefore, in addition to the information of a local group, an eldest local group also knows: *(i)* its parent group, *(ii)* its exhaustion and termination notifiers.

The key point of the distributed implementation is that brethren groups interact as groups in parenthood relationship except that they use implementation-defined notifiers instead of user-defined ones. When a younger local group runs out of energy, it asks its elder brother for more energy. When a younger local group terminates, i.e. when it has no subgroups, no younger brothers, and no tasks, it posts a notification (accompanied with its remaining energy) to its elder brother which then removes it from its younger brothers. When an eldest local group terminates, it reacts as previously described with respect to its parent group. The difficult case is when an eldest local group runs out of energy, since it should not post a notification to its parent unless all its younger brothers are also out of energy. The solution is then to dry the tanks of the younger brothers (but not the subgroups) to the benefit of the eldest brother. A group has no centralised tank but a collection of local tanks connected by implementation-defined notifiers. These inner transfers of energy are invisible to the user; they may use sophisticated techniques to ensure *load balancing* of energy inside the connected tanks of a group[3]. It is only when this connected tank is empty that the eldest

[3]The same schema may be used to provide each task with its own tank, hereby

local group is allowed to notify its parent of its exhaustion.

3.4 Distributed Implementation

A prototype distributed implementation has been implemented using the library Nexus [13]. Nexus is structured in terms of the following basic abstractions: sites[4], threads, global pointers, and remote service requests. A computation executes on a set of *sites* and consists of a set of *threads*, each executing in an address space. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same site.

The *global pointer* (GP) provides a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. A GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between sites by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint.

Practically, an RSR is specified by providing a global pointer, a handler identifier, and a data buffer, in which data are serialised. Issuing an RSR causes the data buffer to be transfered to the site designated by the global pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and pointed specific data are available to the RSR handler. Nexus was ported to the thread package PPCR in order to implement NeXeme, a dialect of Scheme offering a functional version of RSRs [32].

As described in Section 3.3, groups are incarnated by *local groups* on each site; GPs pointing at local groups can be passed between sites. Operations like pause and awake are allowed on remote groups using RSRs. Nexus primitives that use mutual exclusion, like buffer management, were defined so as to prevent preemption by Quantum; also, the thread that polls for incoming messages in Nexus does not run under the sponsorship of a group in order to prevent deadlocks.

We have implemented a weaker version of the semantics for the distributed version. Local groups follow a hierarchical organisation according to a brotherhood relationship. In the current implementation, an exhaustion event is notified whenever a local group has no energy left *without taking into account* the energy available in its younger brothers. For instance in Figure 9, when there is no energy left in g_1 on *Site2*, a notification is run on *Site1* under the sponsorship of g, trying to transfer more energy to g_1. If not enough energy is available in g, a user-specified notification is run in the parent of g. These notifications proceed even though there might be some energy left in the group g_3 on *Site3*. A future implementation will provide a *transparently* distributed notion of group on top of the current system.

3.5 Performance Evaluation

In order to evaluate the performance of the implementation, we wrote several programs whose purpose is to identify the cost of Quantum operations. The code run in our experiment is available from the URL `http://www.ecs.soton.ac.uk/~lavm/Quantum`. We distinguish the shared-memory multi-threaded implementation from the distributed implementation.

The first experiment measures the cost of exhaution notification and group awakening in a shared memory. The program SM-notif creates a group that sponsors a task executing an infinite loop. The group is initially given one unit of energy; its exhaustion handler awakens the group by transfering one unit of energy and also updates a counter indicating the number of exhaustion notifications.

In the current implementation, the infinite thread of SM-notif is suspended as no energy is left in its sponsor, its group supervisor thread is called, which in turn resumes the thread. We compare this operation with an operation for which we intuitively know the cost. In SM-ping-pong, two threads access a critical section *in turn*.

Figure 10 shows the average cost of one operation over 1000 executions. Experiments were run on Indy worstations, Mips R4600, running IRIX 5.3, using 10Mb ethernet. We give the user and system times per operation, and their sum is used for comparison purpose. We see that notification detection and awakening of a group is approximately twice the cost of two threads accessing a critical section in turn.

Remark We use a modified version of the user-level thread package PPCR. Therefore, the time spent is the thread package for scheduling, mutual exclusion, etc. appears in the user time. Also the total time includes the time spent by the Nexus library to poll for incoming messages. □

Program	U (ms)	S (ms)	U+S (ms)	ratio
SM-ping-pong	0.79	0.13	0.92	1.00
SM-notif	1.88	0.31	2.19	2.38
DM-ping-pong	11.62	3.09	14.71	1.00
DM-notif	16.43	2.96	19.39	1.31
SM-pause-awake	12.00	00.80	12.80	1.00
DM-pause-awake	80.02	15.60	95.62	7.47

Figure 10: Performance

The next experiment evaluates the cost of notifications when groups are distributed. Two sites s_1 and s_2 are used in the program DM-notif. An initial group is created on s_1, and a young brother group is created on s_2 by a remote service request; the task sponsored by the young brother immediately consumes all its energy. A notification transfers more energy from s_1 to s_2. Again, we compare the cost of exhaustion detection and awakening of distributed groups with an operation for which we intuitively know the cost. The program DM-ping-pong measure the cost of sending a remote service request to s_2 and of receiving an acknowledgement in the form of a remote service request.

Again, Figure 10 shows the average cost of one operation over 1000 executions. We see that the combined cost of exhaustion detection and energy transfer is only 30% higher than the cost of a communication and its acknowedgement. Also, the distributed memory operation is roughly an order of magnitude more expensive that the shared memory one.

In order to measure the performance, we turned off the sending of messages for the distributed garbage collector, but we maintained garbage collection on each site.

The third experiment evaluates the cost of pausing and awaking groups in a shared memory system. The program SM-pause-awake creates a group g which has two subgroups g_1 and g_2. Each subgroup sponsors a threads that is an infinite loop. The program repeatedly pauses the group g and its subgroups, and awakens them with the same distribution of energy. The results for this experiment varied extremely (as opposed to the others which were rather stable). The reason is that the program does not check that a group hierarchy has been completely awakened before being paused again. It is also

reducing communications with groups.

[4] Nexus distinguishes nodes from contexts; for the purpose of this paper, it suffices to consider that sites are processes.

important to note that the pause primitive creates a new notification thread for each subgroup (as opposed to other notifications which are run by the group supervisor thread created at the same time as the group). This program has therefore a serious effect on memory allocation, and the results displayed are an average over 100 exections.

The distributed variant is DM-pause-awake which involves two sites s_1 and s_2. The group g is created on site s_1 and initiates a remote service request to s_2 which results in a younger brother, sponsoring two subgroup g_1 and g_2. Similary, g and remote subgroups are paused and awaken. We observed that the cost is only 8 times the cost of the shared memory implementation.

4 Discussion and Related Work

Our semantics measure the resources used by computations according to an independent cost model. One might prefer to have other cost models, taking into account, for instance, memory occupancy (duration or size). In order to have a finer control on energy consumption, we are extending $\mathcal{Q}$uantum so as to let the user create and manage *different budgets* from which energy can be consumed; e.g. geographical extent, computation resource, ... Therefore, we can regard energy as any multidimensional (vector) data structure of budgetised resources. Another interesting question is to decide how garbage collection should be billed in a multi-user environment. At the moment, garbage collection is part of the administration cost of the system and is indirectly billed via the cost of allocators.

To the best of our knowledge, three different communities have studied techniques to control computations: the programming languages, the parallel and distributed systems, and the agents communities.

4.1 Engines and Sponsors

Our notion of group is at the intersection of two different ideas: Haynes and Friedman's engines and Kornfeld, Hewitt, and Osborne's sponsors, which we develop below.

Haynes and Friedman [19, 20] introduce the *engine* facility to model timed preemption; variants can also be found in [8, 9, 48]. Engines differ from our groups in a number of ways. Engines are defined in a *sequential* framework and are used to simulate multiprogramming. Since engines do not deal with parallelism, they do not offer control facilities like pause and awake. Another major difference is that a given engine can be executed several times, while a group can only be executed once. Using continuation terminology, engines are "multishot", while groups are "single-shot" [4]. A group is a name and an energy tank for a computation, but, unlike an engine, it does not embody its continuation. Our decision to design "single-shot" groups is motivated as follows. The ability to restart a same computation several times is an unrealistic feature for a distributed language because the computation may be composed of several tasks distributed over the net. Haynes and Friedman also propose *nested engines*, i.e. engines that can create other engines. In their approach, nested engines have the same temporal vision of the world, because each computation consumes ticks, i.e. energy quanta, from *all* parent engines (direct *and* indirect). On the contrary, groups offer a more distributed vision of the world, because groups are tanks, from which local tasks consume energy.

Kornfeld and Hewitt's sponsors [26], Osborne's enhanced version of them [36, 37, 18], and subsequently Queinnec's groups [45, 43], also allow the programmer to control hierarchies of computations in a parallel setting. Osborne's sponsors are entities that give attributes, such as priority, to tasks, which can inherit attributes from several sponsors. A combining rule yields the effective attributes of a task, and then determines the resources allocated to the task. If the group hierarchy changes, priorities should be recomputed, which can be costly, especially in a distributed environment. With $\mathcal{Q}$uantum groups, scheduling of a task is only decided by examining the energy available in its only sponsoring group, which is local. Furthermore, priority is a difficult notion to grasp in a heterogeneous environment, while total work accomplished is more intuitive. In particular, our cost model allows us to program applications searching the best solution at a given amount of energy, i.e. at a given cost. Queinnec's Icsla language has a notion of group which substantially differs from the one presented here. As Icsla is energy-less, pausing a group does not collect energy and can be performed lazily. Also, Icsla does not have any of the notifications of $\mathcal{Q}$uantum. Let us observe that termination notification is a generalisation of `unwind-protect` [49]. Hieb and Dybvig [21] `spawn` operator returns a controller, which can be invoked to suspend or restart part of a computation tree; their approach relies on a notion of partial continuation.

4.2 Parallel and Distributed Computing

Unix-like operating systems offer a different model to control processes: a process is given a unique identifier which can be used to send signals to it, e.g. kill. We believe that this model of control does not provide the appropriate abstraction [37]. Indeed, we want to be able to control computation, but we might not know the processes it is composed of, because the code we execute was not written by us. By associating groups with computations, we abstract from the details of execution. In addition, our model is designed for running in a distributed framework, so that groups can control tasks over different machines.

PVM [14] extends the Unix model to distribution. Each task enrolled in PVM has a unique identifier. PVM has a primitive for sending signals to remote tasks. PVM has also a notion of group, which tasks can dynamically join or leave. Groups are used in PVM for broadcasting messages or to perform synchronisations. MPI-2 [15], the new version of MPI [12], also offers the possibility to send signals to groups of tasks. Some resource management tools [41] deal with memory allocation, CPU allocation, and load balancing. However, we believe that there is no equivalent to the $\mathcal{Q}$uantum facilities in those message-passing libraries, like the ability to resume and suspend computations, or dynamic transfer of resources. This does not mean that such a semantics cannot be programmed on top of message-passing libraries; signalling and broadcasting could be used to implement $\mathcal{Q}$uantum.

The Propero Resource Manager [35] comes as a library to allocate tasks to processors in a multi-processor architecture. In $\mathcal{Q}$uantum, we could regard locations, i.e. processors, as a resource. $\mathcal{Q}$uantum would then offer a high-level interface to Prospero, which would take care of practically assigning processors.

4.3 Agents

The mobile agent community deals with the problem of controlling distributed mobile computations, called *agents*. The most widespread agent systems are Telescript [27, 28], Tacoma [23, 24], Agent-TCL [16], and Ara agents [40, 39]. Very few of them are able to control resources in a similar way as $\mathcal{Q}$uantum.

Telescript [27, 28] is an agent scripting language. Every Telescript agent has a permit that limits its capabilities in the electronic marketplace, like for instance, the right to use a resource, or a maximum overall expenditure of resources measured in *teleclicks*.

Quoting [28], "if the agent exceeds any of its quantitative limits, the engine destroys the agent unceremoniously. No grace period is extended". Our model is more general than the Telescript approach as it allows us to drive computations using pause and awake and to monitor them using asynchronous notifications. Also, our model supports agents that perform parallel and distributed computations, what is usually called multi-agent systems.

Ara agents [40, 39] are equipped with resource accounts called *allowances*, which are similar to our groups [40, p. 6]; as indicated by Peine, they are a recent concept, not complete yet. Several agents can share the same allowance, also called a group. Agents can transfer resources explicitly between accounts. In Ara, agents can be suspended or reactivated: these actions however are performed on agents directly and not on groups [40, p. 23]; as a result hierarchical computation cannot be controlled as in Quantum. In Ara migration operations specify the allowance that an agent should take with it [40, p. 29]. We adopt a different view as our groups are transparently distributed. Ara agents are not notified of an exhaustion [40, p. 35]; Peine consider that this is not a severe problem as an agent may enquire about the existence of a resource. We believe that this argument is not valid in parallel/distributed computing because another parallel computation might consume the resource.

Arthursson *et al.* [1] describe a mobile agent architecture based on the distributed language Erlang. They provide some form of resource control, but an agent is terminated once it has consumed all its resources.

In Agent TCL, an access list specifies the quantity of a resource that an agent can consume; but apparently, no mechanism is provided to transfer new resources to an agent.

Cardelli and Davies [5] present a set of combinators to program the WWW. Some of them deal with real-time operations, such as transfer of data on a medium that can fail. We are studying how Quantum could be extended to support such real-time operations.

5 Conclusion

In this paper, we present the paradigm Quantum, whose purpose is to monitor and control computations in a parallel and distributed framework. Quantum provides a model to control resources in distributed computations, and therefore can be used in agent scripting languages or to program resource administration applications. Besides, it offers the end user the possibility to control hierarchies of computations, to perform some load-balancing, and to introduce some resource-based priority between parallel computations; also, it provides primitives to build "any-resource" algorithms, applying the idea "any-time" algorithms [7] to any form of resource.

As an application, we are building an information-discovery agent [6], whose resources can be controlled and monitored using Quantum primitives. The agent explores HTML pages and is able to move across WWW sites, when following links. The interface to the WWW is provided by a special communication module of Nexus over HTTP. Quantum primitives allow us to delimit the geographic expansion of the agent, following the Time to Live (TTL) approach adopted in network protocols; it is also possible to suspend and resume the agent.

Quantum is sound because it prevents generating energy. Furthermore, it has some features that can be used to enforce security in the application: *(i)* energy cannot be generated, but can only be transferred between computations; all "accounting" operations remain under absolute control of the primitives; *(ii)* groups are the only handle to control computations, and scoping rules of the adopted language guarantee that groups will be visible only where the programmer wishes them to be, *(iii)* there is no primitive that returns the group in which the user code is running, which ensures that user code cannot control its sponsoring group, and hence it cannot control tasks running in parallel with it, unless explicitly passed handles to their sponsoring groups. Security is an important issue in distributed agent-style applications. Using Quantum primitives, there is nothing that prevents users from erroneously transferring resources between groups, or making a group accessible to and then preemptable by another task. However, we believe that some static analysis [50] would be able to detect whether a group might become preempted if made accessible in a public data structure for instance.

Quantum is the core of a consumption-oriented language which is particularly suitable to programming over the Internet. In the future, we plan to investigate a fault-tolerant version of the language, which would be energy aware.

6 Acknowledgements

Thanks to Jonathan Dale for his useful comments. Andreas Kind and Jean-Pierre Briot mentioned a link between Quantum and any-time functions. Luc Moreau was partially supported by EPSRC GR/K30773 and EC project reference ERB 4050 PL 930186, and Christian Queinnec was partially supported by GDR-PRC de Programmation du CNRS and EC project reference ERB 4050 PL 930186.

Bibliography

[1] Johan Arthursson et al. A Platform for Secure Mobile Agents. In *The Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 109–120, April 1997.

[2] Henry Baker and C. Hewitt. The Incremental Garbage Collection of Processes. AI Memo 454, MIT AI Lab, March 1978.

[3] Robert S. Boyer and J. Struther Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the ACM*, 31(3):441–485, July 1984.

[4] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, pages 99–107, Philadelphia, Pennsylvania, May 1996.

[5] Luca Cardelli and R. Davies. Service Combinators for Web Computing. In *Usenix Conference on Domain Specific Languages DSL97*, Santa-Barbara, California, October 1997.

[6] Jonathan Dale and David DeRoure. Towards a Framework for Developing Mobile Agents for Managing Distributed Information Resources. Technical Report M97/1, University of Southampton, February 1997.

[7] T. L. Dean and M. Boddy. An Analysis of Time-Dependent Planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, 1988.

[8] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[9] Michael Eisenberg. *Programming in Scheme*. The Scientific Press, 507 Seaport Court. Redwood City. CA 94063-2731, 1988.

[10] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[11] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995. Technical Reports 238, 239, Rice University, 1994.

[12] Message Passing Interface Forum. A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, June 1995.

[13] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[14] Al. Geist and al. PVM 3 User's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Knoxville, Tennessee, May 1993.

[15] Al Geist, William Gropp, et al. MPI-2: Extending the message-passing interface. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in Lecture Notes in Computer Science, pages 128–135, Lyon, France, August 1996. Springer-Verlag.

[16] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996. http://www.cs.dartmouth.edu/ ∼agent/papers/index.html.

[17] H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[18] Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.

[19] Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24. ACM, 1984.

[20] Christopher T. Haynes and Daniel P. Friedman. Abstracting Timed Preemption with Engines. *Comput. Lang.*, 12(2):109–121, 1987.

[21] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[22] IEEE. *IEEE P1003.1c/D10 Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX)*, September 1994.

[23] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System Version 1. Technical Report 95–23, Department of Computer Science, University of Tromsł, Norway, 1995. http://www.cs.uit.no/ DOS/Tacoma/Publications.html.

[24] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, 1995. Also available as Technical Report TR94-1468, Department of Computer Science, Cornell University. http://www.cs.uit.no/DOS/Tacoma/Publications.html.

[25] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static Analysis of Communication for Asynchronous Concurrent Programming Languages. In *Second International Static Analysis Symposium (SAS'95)*, number 983 in Lecture Notes in Computer Science, pages 225–242. Springer-Verlag, 1995.

[26] William A. Kornfeld and Carl E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[27] General Magic. Telescript Technology: Mobile Agents. http://www.genmagic.com/ Telescript/Whitepapers/ wp4/whitepaper-4.html, 1996.

[28] General Magic. Telescript technology: The foundation for the electronic marketplace. http://www.genmagic.com/ Telescript/Whitepapers/ wp1/whitepaper-1.html, 1996.

[29] Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, Service d'Informatique, Institut Montefiore B28, 4000 Liège, Belgium, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.

[30] Luc Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in Lecture Notes in Computer Science, pages 615–624, Lyon, France, August 1996. Springer-Verlag.

[31] Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 146–156, Philadelphia, Pennsylvania, May 1996. Also in ACM SIGPLAN Notices, 31(6), 1996.

[32] Luc Moreau, David DeRoure, and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 581–590, Passau, Germany, August 1997. Springer-Verlag.

[33] Luc Moreau and Christian Queinnec. Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, California, October 1997.

[34] Luc Moreau and Daniel Ribbens. The Semantics of pcall and fork. In R. Halstead, T. Ito, and C. Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, number 1068 in Lecture Notes in Computer Science, pages 52–77, Beaune, France, October 1995. Springer-Verlag.

[35] B. Clifford Neuman and Santosh Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[36] Randy B. Osborne. Speculative Computation in Multilisp. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 103–137. Springer-Verlag, 1990.

[37] Randy B. Osborne. Speculative Computation in Multilisp. An Overview. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, Nice, France, June 1990.

[38] Patiwat Panurach. Money in Electronic Commerce. *Communications of the ACM*, 39(6):45–50, June 1996.

[39] H. Peine. An Introduction to Mobile Agent Programming and the Ara System. Technical Report ZRI report 1/9, Dept. of Computer Science, University of Kaiserslautern, Germany, 1997. http://www.uni-kl.de/AG-Nehmer/Ara/.

[40] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. of the First International Workshop on Mobile Agents MA'97*, number 1219 in Lecture Notes in Computer Science, Berlin, Germany, April 1997. Springer-Verlag. http://www.uni-kl.de/AG-Nehmer/Ara/.

[41] Jim Pruyne and Miron Livny. Providing Resource Management services to Parallel Applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994. http://www.cs.wisc.edu/condor/publications.html.

[42] Christian Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[43] Christian Queinnec. Sharing Mutable Objects and Controlling Groups of Tasks in a Concurrent and Distributed Language. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, number 700 in Lecture Notes in Computer Science, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[44] Christian Queinnec. DMEROON: a Distributed Class-based Causally-coherent Data Model: Preliminary Report. In *Parallel Symbolic Languages and Systems.*, Beaune, France, October 1995.

[45] Christian Queinnec and David De Roure. Design of a Concurrent and Distributed Language. In A. Agarwal, R. H. Halstead, and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems and Applications*, number 748 in Lecture Notes in Computer Science, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.

[46] Eric Raymond. *The New Hacker's Dictionary*. MIT Press, 1991.

[47] Jonathan Rees and William Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[48] Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, Rice University, Houston, Texas, April 1994.

[49] Guy Lewis Steele, Jr. *Common Lisp. The Language*. Digital Press, second edition, 1990.

[50] Dennis Volpano and Geoffrey Smith. A Type-Based Approach to Program Security. In *Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Lille, France, April 1997. Springer-Verlag.

[51] Mitchell Wand. Continuation-Based Multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[52] M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime Approach to Interoperability. In *ACM Symposium on Operating System Principles*, pages 114–122, December 1989.