*J. King Saud Univ.*, Vol. 16, *Comp. & Info. Sci.*, pp. 65-80 (A.H. 1424/2004)

# Global GAMMA: A Distributed Implementation of GAMMA Using Global Computing

**Salim Ghanemi and Ahmed A. Al Damegh**
*Department of Computer Science, College of Computer & Information Sciences*
*King Saud University, Riyadh, Saudi Arabia*

(Received 23 February 2003; accepted for publication 29 July 2003)

**Abstract.** It is commonly known that writing and debugging parallel programs is more difficult than sequential programs (using traditional imperative or functional programming languages). *Gamma*, which is formalism for programming by multiset transformation, was suggested with the idea that programs are expressed in very abstract way. This makes Gamma very suitable for writing "correct" parallel programs. Gamma is a specification model, which is far away from a real architecture, thus designing reasonably efficient implementation of the language is not straightforward. The main goal of the *GLOBAL GAMMA* project is to develop a parallel distributed implementation of Gamma on the Internet (on a *Global computing* environment), which will exploit the idle time of Internet-connected computers.

**Keywords:** Parallelism; Gamma; Distributed gamma; Chemical reaction model; Multiset transformation; Very high-level languages; Global computing; Distributed Processing.

## 1. Introduction

The rapid penetration of computers into commerce, science, and education owed much to the early standardization on a single machine model, the *von Neumann* computer. The von Neumann machine model assumes a processor to be able to execute "sequences" of instructions. While it is possible to program a computer in terms of this basic model by writing machine language, this method is for most purposes prohibitively complex. Hence, modular design techniques are applied, and higher-level programming languages are developed.

Parallelism makes programming task more difficult in terms of writing and debugging programs especially using traditional imperative or functional programming languages, because the programmer has to mentally manage several threads of control [1].

J-P. Banatre et al proposed Gamma [1] in 1986, a formalism for programming by multiset transformation, in which programs are expressed in very abstract way, leaving any artificial sequentiality and details. This makes it very suitable for parallel programming. Also due to its abstractness, abstract specifications of programs can easily be transformed into Gamma programs, which lead to easy construction of "correct" parallel programs [5].

Gamma is a specification language, thus a Gamma program is far away from a real architecture, and designing reasonably efficient implementation of the language is not straightforward. Several attempts have been made to the implementation of Gamma on different platforms (multiprocessor computers, network of workstations...) but, as far as we know, no implementation of Gamma introduced to utilize the power of the huge number of Internet-connected computers as one computational resource. But on the other hand, there have been several studies on *Global computing* [7]. Global computing systems harvest the idle time of Internet connected computers which may be widely distributed across the world, to run a very large and distributed application.

## 1.1 Project goal

The main goal of this project is to develop a parallel implementation of Gamma on the Internet or on a Global-computing infrastructure. The idea is to provide programmers with an environment to write correct Gamma programs and have these programs executed in an implicitly parallel way on the underlying Global computing infrastructure, without having the programmer to care about this parallelism of execution.

The underlying Global computing infrastructure used is constituted from computers of volunteers who have the interest to join the Global-computing environment. The solution should ensure easy and secure participation of volunteers. In short, we seek utilizing Global computing techniques to derive a parallel-distributed implementation of Gamma. So, we will refer to this project as *"GLOBAL GAMMA"*.

## 1.2 Paper organization

The remainder of this paper is organized as follows: in section 2 we introduce the Gamma formalism. Section 3 provides a review of Global Computing Systems. *GLOBAL GAMMA* specifications are highlighted in section 4. In section 5 we give a high level design of *GLOBAL GAMMA*.

## 2. Programming with Gamma

### 2.1 Introduction

Gamma [1] (General Abstract Model for Multiset mAnipulation) is a kernel language for describing programs in terms of multiset transformation. It is a high-level language that has been introduced intuitively from the chemical reaction metaphor. It allows the description of programs in a very abstract way without introducing unnecessary sequentiality [5], i.e., sequentiality that is not required by program logic. Also it is based on the distinction between the correctness and efficiency of programs, with correctness being the primary concern in program development. To illustrate this, let us consider the example of finding the maximum element of a set. To solve this problem using an imperative language, the set can be represented as an array a[1..n] and the program is

```
m := a[1];
i := 1;
while i < n
  { i := i + 1 ;
    m := max (m, a[i]) ; }
```

While the condition $i < n$ holds, index $i$ is incremented and $m$ is computed. This solution imposes a strict ordering between comparisons of elements, the first is compared with the second, and then their maximum is compared with the third...and so on.

Using Gamma the solution can be obtained by performing the comparisons with any order and in an abstract form. A possible Gamma program can be:

```
Max(s): Γ((R,A)) (s) where
               R(x,y) = x ≤ y
               A(x,y) = {y}
```

Meaning, while there are at least two elements, select two elements, compare them, and remove the smaller one.

A Gamma program is a pair of *reaction condition* $R$ and *action* $A$. Execution proceeds by replacing the multiset elements satisfying the condition $R$ by the product of the application of the action $A$. This is analogous to chemical reactions as the set being the chemical solution, $R$ being the reaction condition to be satisfied by the reacting elements, and $A$ describes the result of the reaction. The computation terminates when a stable state is reached, i.e., when no elements in the set satisfy the reaction condition.

In case of several disjoint pairs of elements satisfy the condition; comparisons and replacements can be done in parallel. Thus, Gamma involves implicit parallelism. The only data structure used in Gamma is the *multiset* or the *bag*. The essential feature of multisets is that data is no longer seen as a hierarchy that needs to be walked through or decomposed in order to extract atomic values. Instead, atomic values are gathered into one single bag and the computation is the result of their interactions.

Another feature of Gamma is the *locality property*: independent elements can interact with each other and produce new elements in completely independent (or simultaneous) way. Hence, a reaction condition should not include any global condition on the multiset, such as properties on the cardinality of the multiset [2]. This property is the basic reason why Gamma programs do generally exhibit a lot of potential parallelism [1, 4].

## 2.2 Gamma programming style

In Gamma, a program is no longer a sequence of instructions modifying a state, or a function applied to its arguments but rather a multiset transformer operating on all the data at once. The development of Gamma programs entails the choice in data representation and the choice in the type of transformation applied to this data.

Let us consider the problem of prime number generation. The task is to produce all prime numbers less than a given *N*. The solution is based on the fact that the $(i+1)^{th}$ prime ($i \geq 1$) is the smallest integer exceeding the $i^{th}$ prime that is not divisible by the first $i$ primes. Integers greater than $\sqrt{N}$ that have not been eliminated by divisions by integers less than √N are primes [1].
A Gamma solution is:

```
PrimeNumbers(N) = Γ((R,A))({2,...,N}) where
                  R(x,y) = multiple(x,y)
                  A(x,y) = {y}
```

Here, multiple ($x$, $y$) determines weather $x$ is a multiple of $y$ or divisible by $y$.

Imperative solutions to this problem are intricate, because they give a detailed description of the execution order. But the Gamma solution is easier and concise, because computational details are left unspecified.

### 2.2.1 Data decomposition

The first step for designing a Gamma program is to find a suitable representation of data as a multiset, since the multiset is the only data structure. If the program, for example, has to operate on basic values, such as integers, a suitable decomposition of these values has to be found. The prime numbers example above decomposes its argument *N* into a multiset {2...N}. Complex data such as sequences, trees or graphs can be flattened into multisets in a straightforward way. For example, a sequence can be decomposed in a multiset of (*index*, *value*) pairs. Flat multisets make all the components of the data structure directly accessible, regardless of their position in the structure. Gamma has a *topological view of data types* in contrast to the *recursive view of data type* where a walk through is necessary to access a particular component [1].

### 2.2.2 Relaxation

Relaxation is a method used to solve problems in an iterative way. First, an initial multiset is produced as an estimate of the solution, and then a series of actions used to decrease and "relax" errors in the initial estimate until a proper solution is found. In the Prime Number example, the initial multiset is $\{2,\ldots,N\}$, which is a rough estimate of primes less than $N$, and the computation continues to remove non-primes.

### 2.2.3 Data expansion and reduction

Data expansion and reduction are two other Gamma programming techniques. Data expansion means the decomposition of multiset elements into a collection of items. Computation stops when the multiset is a collection of indivisible elements. Data reduction, on the other hand, corresponds to the case where a multiset of items is reduced to a singleton by successive application of the action. The following program explains these two concepts. It computes *Fibonacci* function, which is defined as:

*Fibonacci*($n$) = **if** $n \leq 1$ then 1 **else** *Fibonacci*($n - 1$) + *Fibonacci*($n - 2$)

```
Fib(n)   = m where
           {m} = sigma(gen({n}))

gen(n)   = Γ((R1, A1),(R2, A2))(n) where
           R1(n)  = n > 1
           A1(n)  = {n - 1,n - 2}
           R2(0)  = true
           A2(0)  = {1}
sigma(M) = Γ(R, A)(M) where
           R(x, y)  = true
           A(x, y)  = {x + y}
```

*gen*($n$) in the above example performs a data expansion, multiset $\{n\}$ is expanded to a multiset of indivisible elements, which are ones. *sigma*($M$) performs a data reduction, the multiset $\{1,\ldots,1\}$ is transformed into a singleton multiset, which is the value of *Fibonacci*($n$), by a series of sums.

Relaxation, compared to data expansion and data reduction, does not transform the structure of the multiset but proceeds by successive refinements. Those three programming techniques are the building blocks for writing Gamma programs.

## 2.3 More examples

In this section we give more examples to show the expressivity of Gamma and its ability to accommodate different types of problems. We refer the reader to [1] for more examples.

### 2.3.1 Factorial

The following program computes *n!*:

```
fact(n) = Γ((R,A)) ({1, ... ,n}) where
```

```
        R(x, y) = true
        A(x, y) = {x * y}
```

Here, the reduction technique is used to solve the problem. Here also we stress on the fact that no constraint is put on the order of multiplication.

### 2.3.2 Sorting

To sort elements of an array in ascending order, we use a multiset of pairs (*index, value*) and the program exchanges ill-ordered values until all values are well-ordered.

```
    Sort(array) = Γ((R, A)) (array)  where
        R((i, v),(j, w))=(i > j) and (v < w)
        A((i, v),(j, w))={(i, w), (j, v)}.
```

### 2.3.3 The majority element

The majority element of a multiset $M$ is an element occurring more than $|M|/2$ times in the multiset. Assuming the solution exists, the following program finds it:

```
    MajElem(M) = Γ(R, A)(M) where
        R(x, y) = x ≠ y
        A(x, y) = {}
```

The computation proceeds by having non-equivalent elements cancel each other out. The resulting multiset will have at least one instance of the majority element.

## 3. Global Computing Systems

### 3.1 Introduction

The key idea of *Global Computing* is to harvest the idle time of Internet connected computers which may be widely distributed across the world, to run a very large and distributed application. All the computing power is provided by volunteer computers, which offer some of their idle time to execute a piece of the application. Thus Global Computing extends the cycle stealing model across the Internet [3, 6, 7, 11].

Indeed Internet is pervasive and the number of connected devices is constantly growing. The challenge is to exploit the so many unused resources to build a very large parallel computer [8] or a *global computer*.

Due to the limitation in network performance, GCS target mainly applications that can be broken into coarse grain tasks, either independent or scarcely communicating [7]. By going back to Gamma, we see that this requirement is satisfied by Gamma model due to the locality property described earlier.

## 3.2 Global computing issues:

### 3.2.1 GCS architecture

A GCS is logically organized as a 3-tier system, see Fig. 1. The *request* layer submits a job. The *broker* layer marshals the request, then maps and schedules work, and the *service* layer actually computes. While the 3-tier organization is fairly common, GCS have two major originalities. First, the logical architecture does not have to map exactly the physical one, i.e., requesting, servicing and ultimately brokering (the logical architecture) are provided by the same resources (the physical architecture), the Internet links and the collaborating computers. Second, the resources are highly volatile and users untrustworthy. Computers can come and go freely, and the same is true for users; the bandwidth, latency and security of Internet connections is highly variable [7].

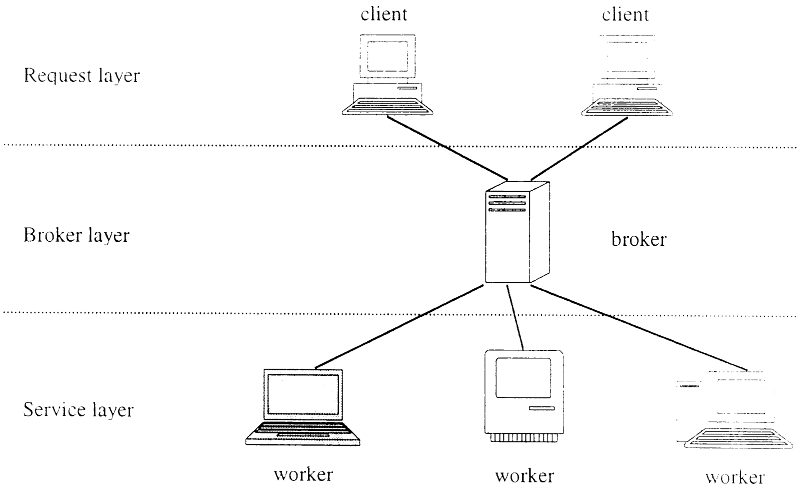

**Fig. 1. GCS architecture.**

### 3.2.2 Performance

Scheduling is the key for application performance in the context of a GCS environment.

Static information is inadequate for the development of efficient schedulers. The obvious reason is that a Global Computer is essentially a shared resource, with external (with respect to the scheduler) users of the computing power and externally generated network traffic. Dynamic information on all resources of the Global Computer must be embodied in the performance-modeling scheme, so as to provide forecasts that are one of the inputs of adaptive schedulers [7].

Predictions about processor and network workload will be used to map and schedule the tasks on a GCS. Due to the long-term unpredictable nature of this system, scheduling should be a dynamic process iterated many times as external conditions change. Scalability of the scheduling algorithm itself is a main concern, both with respect to the number of tasks and the size of the GCS. Two possible alternative can happen: If the number of resource shrinks, then we have to do with what is currently available and hope that this situation will change in the near future. However, if the number of resources increases then it means that our solution will be reached faster than predicted.

### 3.2.3 Fault tolerance

GCS largely stretch the concept of Fault Tolerance. The issue becomes how to compute efficiently in an environment where faults are normal, not exceptional, events [7]. Fault tolerance is an issue both at the broker and the service level. When physically distributed, the broker level should maintain a consistent view of a distributed data space, which is a classical problem. Full P2P systems devoted to file storage and retrieval have implemented broker fault-tolerance based on redundancy.

At the worker (service) level, a GCS has to ensure that the computation will make some progress, as long as functional resources are available. However, defining what is a functional resource is somehow blurred in such systems. The most traditional way is to consider only *online* resources, that is, workers currently registered in the GCS. As soon as they do no appear as registered anymore (they go offline), they are declared as faulty and they are not functional. Thus, their assigned task(s) is lost, and it must be restarted from scratch, except for checkpointing. In another way, one may consider that resources come and go, and that it does not make sense to base the policy on them, but only of the tasks to perform. If there exists a registration system, this allows computations to carry on *offline*, when a computer is technically faulty.

With offline systems, the problem is now how to deal with truly faulty (never coming back) resources. One solution has been proposed in the framework of parallel computing, with the concept of eager scheduling [7]. Eager scheduling is not a specific scheduling policy, but a layer over such policy. When all available work has been assigned, unfinished tasks are re-assigned to workers, which become idle. This principle has been implemented in Javelin [10].

Another important issue is long-running tasks. To guarantee their progress in a volatile environment, some form of *checkpointing* [7] must be implemented. In essence, checkpointing is any technique that allows for saving the state of the computation so as to restart it from the reached point. Long-running applications generally include a simple form of checkpointing through files. In the online scheme, these files should be saved through the network, while in the offline scheme they could be written only locally.

### 3.2.4 Security

In the P2P scheme, workers will run completely untrusted code, which may be malicious or erroneous. Encryption techniques, such as SSL, provide reliable data and code transmission, but this is only a small, if mandatory, step, to security. Moreover, the privacy of the host running the worker must be guaranteed. This level of protection requires the worker to be run in an environment that isolates it from the physical host resources.

Java has been the first integrated and modular sandboxing solution. A pointer-safe language executed in a virtual machine with extended dynamic type and array-bound checking protects against malicious or erroneous use of processor resources. Security models allow limiting access to the network and peripherals (displays and files), in a configurable way. Since Java 1.2 many Global Computing projects have used the most secure Java framework, namely Applets [9, 10], or Java applications with adequately configured security policy.

## 4. GLOBAL GAMMA Specifications

As mentioned in section 2, due to its abstractness, it is easy to construct correct parallel programs using Gamma from abstract specifications of programs. Also, Gamma programs exhibit a lot of potential parallelism and our goal of this project is to develop a parallel implementation of those Gamma programs on Internet-connected computers (or a global computing system) utilizing their idle time.

In the followingو we highlight the specifications that GLOBAL GAMMA should meet.

- **Gamma development environment:** An environment should be developed for the programmer to write and run Gamma programs. He should be able to declare and initialize multisets, as well as, defining reaction conditions and actions. The syntax used should be close to the syntax of the basic Gamma presented in section 2 as possible.
- **Compile and run Gamma programs:** When the programmer finishes writing his Gamma program, he should be able to easily compile the program and then have it executed in parallel on the underlying global computing environment in a transparent way, as if he uses a single machine.
- **Presentation of results:** After the completion of the computation, the resultant multiset is presented to the programmer and may be some other statistical information, such as execution time; number of reactions took place...etc.

- **Easy volunteer participation method**: The underlying global computing environment is constituted from volunteers' machines (or workers). So, it should be easy for the volunteer to contribute with his machine.
- **Security:** One important requirement is ensuring security of hosts. Workers should run safe code that will not lead to worker crashes.
- **Scalability**: Hosts come and go to and from the system, thus, the system should be able to accommodate any number of participating hosts and should dynamically adjust it self to variations of hosts numbers.
- **Fault-tolerance:** As noticed, a fault in a global computing environment is a rule not an exception. A host can fail at any moment. The system should be able to determine faulty hosts and recover from this fault by reassigning that host's task to another free host.

## 5. Global GAMMA Design

In this section we give the overall design of *GLOBAL GAMMA* that should accommodate the requirements. Fig. 2 shows the overall *GLOBAL GAMMA* structure. It consists of four layers as generally explained below.

The requirements dictates having a *Gamma programming environment*, which will be used by the programmer to write, compile, and run Gamma programs, as well as presenting computation results to him.

A *Gamma Machine* should take the compiled Gamma program and execute it. It will be responsible for initializing and maintaining the multiset. By maintaining the multiset we mean removing and adding elements to the multiset. It is also has the responsibility for choosing elements for reactions (computations), and the detection of computation termination. It works as the request layer in a GCS.

*Global Computer* layer, acts like broker layer in GCSs. It maintains a list of registered computers (hosts) that are willing to run computations. It receives computation tasks from the Gamma Machine and assigns them to workers.

Worker computers constitute the service layer and run the *Gamma program*, which does the computation on worker computers. It receives tasks from Global Computer, executes them, and sends the results back to the Global Computer.

### 5.1. Gamma program

The input to the system is a Gamma program written by the programmer in the Gamma Programming Environment. The Gamma program consists of four sections:

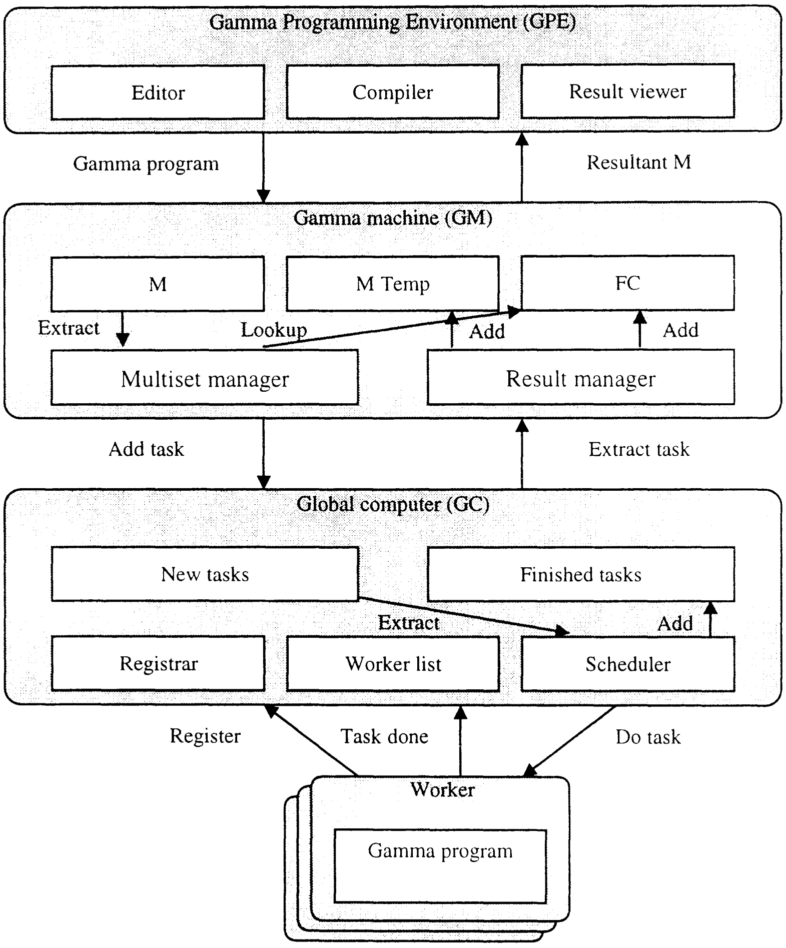

**Fig. 2 Global GAMMA structure.**

1. **Multiset element type definition**: a multiset element can be either simple, like an integer (as in Max example shown earlier) or complex tuple of the form ($v_1$, $v_2$, ...$v_n$) as (value, index) pair of Sort example. Element type will be defined as a tuple of (*value*: *type*) pairs, as follows:
   `Element Type = ( value`$_1$`: type`$_1$`; value`$_2$`: type`$_2$`; ...; value`$_n$`: type`$_n$`)`
   In the Sort example, the element type definition will be:
   `Element Type = ( value: real; index: integer)`

2. **Multiset initialization**: a multiset initially empty. The programmer needs to specify how it will be initialized in the multiset initialization section.
3. **Reaction condition**: a $k$-ary reaction condition. By $k$-ary we mean a condition that tests a number of $k$ elements, or takes $k$ elements as its arguments. Reaction condition will be declared in a similar way like the examples shown in chapter 2.
4. **Reaction Action**: Also a $k$-ary action specifying the transformation needs to be applied to the multiset.

In the current design we will consider only Gamma programs with only one reaction condition and action pair.

Java will be used to internally represent Gamma programs. The selection of Java is due to its interoperability and security model. When a Gamma program is compiled, an intermediate Java code is generated which then gets compiled to Java byte code by a Java compiler.

The reaction condition and action code is compiled along with other control code into a Java applet, which is the Gamma Client that will be executed by workers.

## 5.2. Gamma program execution

The execution of a Gamma program proceeds in three stages: initialization, computation, and termination.
In the following subsections we illustrate generally what happens at each stage.

### 5.2.1 Initialization

Once the Gamma program is compiled and ready for execution, Gamma Machine creates and initializes the multiset. At the other end of the system, workers download Gamma Client and register to the Global Computer.

### 5.2.2 Computation

During this stage, the computation or multiset transformation takes place. The computation will proceed as follows:

1. *Multiset manager* randomly extracts $k$ elements from multiset $M$ that do not form a *Failed Combination*, i.e., the same set of $k$ elements has not previously tested against the reaction condition $R$ and failed. Gamma Machine keeps track of failed combinations of elements in *FC* list.
2. The $k$ elements are packed into a *Task* object and passed to Global Computer for computation by adding it to the *New Tasks* queue.

3. Multiset manager continues extracting $k$-combination of elements and passes them to Global Computer until ($|M| < k$), or there are no more elements that can constitute a computation task.
4. Global Computer in its turn schedules those computation tasks according to the available workers. It keeps track of assigned and unassigned tasks. Whenever a worker joins the environment or finishes its task it is assigned a new task. Faulty workers need to be determined and their pending tasks are reassigned to free online hosts.
5. When a worker receives a computation task (or $k$ elements ($x_1$, $x_2$,...,$x_k$)), it proceeds by testing $R(x_1, x_2,\ldots,x_k)$ trying all possible $k!$ permutations of elements $x_1, x_2,\ldots,x_k$ one by one until:
    a. $R(x_1, x_2,\ldots,x_k)$ is satisfied: in this case the action $A(x_1, x_2,\ldots,x_k)$ is applied on the elements in the same order as they appear in $R(x_1, x_2,\ldots,x_k)$
    b. All $k!$ permutations are exhausted with no one satisfying $R(x_1, x_2,\ldots,x_k)$
6. as a result of the application of the action, elements are removed from $M$, returned back to $M$, or new elements are generated and added to $M$. In case of (a) the worker will tag the resulting elements as *removed*, *returned*, or *new* then sends them back to the Global Computer and indicates that the reaction took place.
   In case of (b), all elements are tagged as *returned* by the worker and sent back to the Global Computer. The worker also indicates that the combination of elements failed to react.
7. Global Computer passes the results back to the Gamma Machine by placing them into the *finished tasks* queue which in its turn does the following:
    a. In case of successful reaction (case (a) above):
        i. *returned* and *new* elements added to *MTemp* (a temporary multiset)
        ii. *removed* elements are discarded and all the failed combinations in *FC* that reference them are deleted.
    b. In case of failed combination (case (b)):
        i. *returned* elements are added back to *MTemp*.
        ii. their combination is added to *FC*. So, it is not selected for computation again.
8. when the number of elements in $M$ becomes less than $k$ (or $|M|<k$) Gamma Machine engine move elements from *MTemp* into $M$, or $M = M \cup MTemp$, and continues computations.

### 5.2.3 Termination

In Gamma the reaction continues until a stable condition is reached, i.e., when all elements (or all combinations of elements) do not satisfy the reaction condition.

In the context of *GLOBAL GAMMA* this happens when

i. $\binom{|M|}{k} = |FC|$ ; and

ii. no elements are out of $M$ for computation.

This means that all the combinations of elements of $M$ are failed combinations, because the number of all possible combinations of elements of $M$ ($\binom{|M|}{k}$) equals the number of failed combinations in $FC$ ($|FC|$), considering that $FC$ always holds all failed combinations of *live* elements in $M$. by live we mean elements that have not been removed from $M$.

Part (ii) of the termination condition above ensures that all elements are in $M$ and there are no elements that are out of $M$ participating in a reaction. Or in other words, the Global Computer scheduler has no in-progress or unassigned tasks.

When termination is detected the resulting multiset $M$ is passed to Gamma Programming Environment and the programmer is provided with the result.

## 6. Conclusion

In this paper, we mainly discussed some implementation issues for the Gamma Formalism using the globally available processing power as termed by the Global Computing. Many similar works are found on the literature. They all used conventional parallel machines with limited processing power but none have attempted to use the global computing. We believe that the huge available processing power would give better performance figure, in the sense that most of the possible reactions could be tested in parallel. At this stage, we are in the process of implementing the Global Gamma. Another interesting idea that we found is the possibility of testing all the possible conditions at once, each on a different processor. We are exploring this direction by having each processor generating one possible permutation in O(1).

## References

[1] Jean-Pierre and Banâtre, Daniel Le Métayer. "Programming by Multiset Transformation." *CACM 36*, No.1 (1993), 98-111.

[2] Jean-Pierre Banâtre, Pascal Fradet and Daniel Le Métayer. *"Gamma and the Chemical Reaction Model: Fifteen Years After."* WMP 2000: 17-44.

[3] Almasi, G. and Gottlieb, A. *Highly Parallel Computing*. 2nd ed. The Benjamin/Cummings Publishing (1994).

[4] Pascal Fradet and Daniel Le Métayer. "Structured Gamma." *Science of Computer Programming* 31, No. 2-3 (1998), 263-289.

[5] Gladitz, Katia and Kuchen, Herbert. "Shared Memory Implementation of the Gamma-Operation," *JSC* 21, No. 4 (1996), 577-591.

[6] Hong, Lin. *"Chemical Reaction Model Based Parallel Programming: Synthesis, Semantics, and Implementation."* PhD Thesis, University of Science and Technology of China (1997).

[7] Germain, Cécile, Fedak. Gille, Néri, Vincent and Cappello, Franck. "Global Computing Systems." LSSC (2001), 218-227.

[8] Germain, Cécile, Néri. Vincent, Fedak, Gille and Cappello, Franck. *"XtremWeb: Building an Experimental Platform for Global Computing."* GRID (2000), 91-101.

[9] Nisan, Noam, London, Shmulik, Regev, Oded and Camiel, Noam. *"Globally Distributed Computation over the Internet - The POPCORN Project."* ICDCS (1998), 592-601.

[10] Christiansen, Bernd O., Cappello, Peter, Ionescu, Mihai F., Neary, Michael O., Schauser, Daniel Wu, Klaus E. "Javelin: Internet-based Parallel Computing using Java." *Concurrency - Practice and Experience* 9, No.11 (1997),1139-1160.

[11] Alexandrov, A. D., Ibel, M., Schauser, K. E. and Scheiman, C. J. "SuperWeb: Towards a Global Web-based Parallel Computing Infrastructure." *11th International Parallel Processing Symposium* (IPPS'97), Geneva, April 1997.

# جاما العالمية : تطوير متوازي وموزع لجاما باستخدام الحوسبة العالمية

**سليم غانمي و أحمد الدامغ**

*قسم علوم الحاسب، كلية علوم الحاسب والمعلومات، جامعة الملك سعود، ص.ب ٥١١٧٨،*
*الرياض ١١٥٤٣، المملكة العربية السعودية*

(قدم للنشر في ٢٠٠٣/٠٢/٢٣م؛ وقبل للنشر في ٢٠٠٣/٠٧/٢٩م)

**ملخص البحث**. من المتعارف عليه إن كتابة و تصحيح البرامج المتوازية وتصحيحها أكثر صعوبة و تعقيدا من كتابة و تصحيح البرامج التسلسلية التقليدية، وذلك عند استخدام لغات البرمجة التقليدية. جاما، أو النموذج المجرد العام لتحويل المجموعات المتعددة (GAMMA) هو مفهوم تم طرحه لكتابة البرامج بشكل عالي التجريد خال من تفاصيل (و تسلسل) التنفيذ لهذه البرامج. هذا التجريد العالي في وصف البرامج يجعل من جاما وسيلة مناسبة لكتابة برامج متوازية صحيحة.

جاما، عبارة عن نموذج وصفي لايوجد له بنية حاسبية مقابلة لتطبيقه، على عكس نموذج فون نيومان التسلسلي الذي يقابله بنية الحاسبات التقليدية، لهذا السبب فإن تصميم نموذج لتطبيق جاما ليس بالأمر السهل.

الهدف الرئيس لهذا المشروع هو تطوير تطبيق متوازي و موزع لجاما باستخدام الشبكة العالمية لاستغلال الوقت المهدر للحاسبات المتصلة بالإنترنت.