

---

# The Eximius Distributed Processor

---

**S. Kazadi**

Jisan Research Institute  
28 North Oak Avenue  
Pasadena, CA 91107  
USA  
sanza@jisan.org

## Abstract

This paper describes the Eximius distributed processor. The Eximius processor is a virtual processing platform which utilizes swarm engineering techniques in the maintenance of a computational swarm of processors. Under 700Kb in size, the processor has the capability to establish and restore communication with a virtually unlimited number of potential computational partners, as well as to draw new partners into the swarm at an exponentially increasing rate of information propagation. Any processor can utilize the power of other processors, with partitioning of processing time and addition and subtraction of new processors occurring seamlessly. We illustrate the capability of the processor on a small problem initiated from multiple machines of differing platforms and capabilities.

**keywords:** swarm engineering, Eximius, distributed computing

## 1 Distributed Computing

With the advent of a virtually global internet bolstered by rapid advances in speed of computing systems, the potential for completely new computing systems was ushered in. These systems [2][3][4][5], which are decentralized by nature, have the potential to allow the creation of completely new computational systems which operate under entirely different rules and have extraordinary capabilities. The simplest such systems center around the use of computational resources that exist on other computer systems. These are the peer-to-peer systems in which computational resources on machines outside of the user's machine can be utilized to accomplish computational tasks. The more complex systems, stemming from parallel computing, actually take large computational tasks, break them into pieces, and have these pieces completed on multiple machines of possibly different platforms and merge the outcomes of each of these processes, completing the larger overall computation.

What has yet to be accomplished is the development of operating systems that allow for the secure distributed computation of large numbers of independent calculations in a way that allows unlimited accessibility both to

software developers and to software users. The method one might search for should be capable of being overlaid on any software platform, allow any user to become part of the greater distributed environment, require little hardware overhead, allow for the development of an unlimited amount and generality of software, and be dynamic in the sense that new users and existing users can be added to or seamlessly removed from (for instance in the case of network failure) the overall group, leaving computations both intact and correct.

In this paper, we explore the design of the Eximius distributed processor, which accomplishes these design requirements. In Section 2, we will explore the motivations for the design specifications of this processor. In Section 3, we describe the design specifications of the system. Section 4 examines the performance of the processor on a small network of computers of limited computational power. Specifically, we demonstrate the multitasking capability of the network and demonstrate the ability of the network to significantly extend the capabilities of very slow computers in our network. Section 5 provides some discussion and concluding remarks

## 2 Swarm Engineering and the design of Eximius

Swarm engineering is the main guiding design paradigm behind the design and implementation of Eximius. Swarm engineering centers around the development of minimal conditions required in multiagent systems required to achieve a global outcome using multiagent systems. This field finds its origins in swarm intelligence, and specifically centers around the use of swarms in the completion of complex high level tasks using groups of simple agents. The design of the overall Eximius system is motivated by high level goals whose basic design guides the processor design.

## 2.1 Swarm Engineering goals for the Eximius system

There are several overall global goals of the Eximius processor system. The first is that the system should be decentralized. This allows the system to be used by any user or group of users, with small local swarms and large global swarms, with no change of local structure or organization. Individuals should be able to join easily and leave easily, without disruption of any computations currently under way in the swarm. Network changes, local or widespread failures, widespread or local additions to the network should become seamlessly dealt with. Neither the swarm utilizing a computational resource nor the resource itself should suffer failures in computations already underway if individual failures occur.

The second global feature required in the design of the processor is the ability to equipartition the swarm based on the particular processes running across the network. That is, if there are  $N$  processors on the network, and there are  $M < N$  processes running on these processors, then the design will be optimal if the average relative computation of each of the processes is, at least in the large number limit, equal to

$$\tau \approx \frac{\tau_c}{M} \quad (1)$$

where  $\tau_c$  is the time of the computation when using the entire network. This second global feature is important in allowing multiple users to take advantage of the large distributed processing power of the overall system. However, our goal is to make this happen in a completely decentralized way so as to require no global controller or supervisory agency.

Our third feature is the design of a system that efficiently uses the computational resources in its repertoire. That is, if each computational resource has a speed in any unit system given by  $s_i$  where the indices  $i$  run over the entire range of machines in the swarm, the total computations involved in the overall computation will partition according to

$$T = \alpha \sum_{i=1}^N \frac{1}{s_i} \quad (2)$$

for some constant  $\alpha$  and total number of computations  $T$ . This is the most efficient use of computational resources. Our swarm should be designed in such a way that if this is not strictly obeyed, it is obeyed in the large computation limit.

Our fourth and final global feature is the design of a system that is secure, both in terms of the programs running across the swarm and in terms of the data systems upon which it is running. Of primary concern to many distributed system designers is the design of systems that

allow distributed processing, but do not compromise the systems that make up the swarm. Thus, the system must be secure in the sense that it is not possible to use the system to breach the security of any element of the overall system. While this cannot be strictly the case, since computer systems that are not currently known to a potential hacker may become known to the hacker through the use of the overall swarm, we limit the danger to that afforded by a secondary attack, rather than a primary attack using the swarm. Moreover, we are concerned with the design of a system which is both capable of protecting data being sent back and forth between different parts of the system and capable of verifiably guaranteeing the data that is coming from the system. Data computed in remote locations can be subject to faulty computations, corrupt data transmissions, and deliberate sabotage. Methods of dealing with these issues must be a concern when putting together a swarm of independent computational centers.

Thus, the goal of the swarm design is the production of a swarm of independent computational agents which has a number of adaptive properties, computational properties, and security properties. The first step in the design of such a system is, of course, the development of the provable minimal design requirement of such a system.

## 2.2 Provable minimal design requirement

In order to motivate the minimal design requirements, we must first provide a model of the system. We may assume that the system, as is probably the simplest case in terms of implementation, is made up of a set of independent computational units. Each unit may be represented by  $u_i$  where  $1 \leq i \leq N$ . Moreover, we suppose that each unit has a speed  $s_i$  in some unit system. Finally, we assume that each computation  $C$  is made up of  $K$  discrete pieces  $c_k$  such that each element must be computed on a single processor.

### 2.2.1 Decentralized Design

Our first requirement is that the software and the actions taken by the software on all members of the swarm be identical. This allows any processes that may occur on any one given machine to be able to occur on any other machine. Thus, if one machine served as a temporary center of computation, then this would indicate that any other machine could serve as a center of computation as well. This requirement removes the possibility of having a permanent central processor.

### 2.2.2 Optimal Processor Use

We assume that each processor is aware of all other processors on the network, either directly or by proxy. In this case, all processors may be contacted by the originating processor by sending messages to all members of the group. Let us define a *request sweep* as a set of messages to all known potentially idle machines which request computation with a given set of parameters (which may be minimal speed of the processor, maximal time of computation, maximal number of jumps on a network, etc.). A *potentially idle* processor is a processor that is not known to be active. We assume that any process that requires more computation will initiate a request sweep, and all idle processors willing to take on the computation will be contacted and able to respond according to their current state. We assume that a computation request will be terminated when enough processors have responded affirmatively to the computation request.

The minimal *negotiated* recruitment between any two processors requires two steps. In step one, an initial request is made to a machine whose state is unknown. This request must contain enough information for the negotiation to be complete, which indicates that the request has all requisite information contained within. The request may also contain the computation, which amounts to a set of data and the steps involved in the computation<sup>1</sup>. The second step involves the acknowledgement of the commencement of computation on the machine in question.

This minimal negotiated recruitment has two immediate flaws. First, in adding the computation to the original request, the requesting machine adds what might be a large amount of new network traffic to the network for what might amount to an unsuccessful attempt to recruit computation. The second is that in order to request computation of multiple pieces of the larger computation, it is necessary to receive a positive answer from some machine before moving on to a second computation. The time required for the initial message and the second message may be large compared to the time required for the initial message.

In view of these difficulties, we have implemented a three-part minimal procedure for recruiting computation which satisfies the requirements that network traffic is minimized, and the number of computations serviced per recruitment message is maximized. In the event that a single message is sent out and only one possible reply can come back, the maximum ratio is one. However, if requests can be made by proxy to large numbers of

<sup>1</sup>This last step may be avoided if the computation takes place on a machine which already contains the code. However, in general, this may not be the case.

processors, this ratio may be much greater than one.

The minimal protocol consists of three steps. In the first step, a minimal message is sent to each known processor requesting computation. The message contains the minimal information set required to accomplish the negotiation. During the second step, those processors for which the negotiation is successful may respond with a minimal message that indicates their willingness to take the computation with the negotiated parameters. During the third step, any available computations will be uploaded to the processor which responded positively.

We also assume that there is a delay  $\gamma$  between successive request sweeps.

Let us now formalize the expected computation time. Suppose that there are  $N$  machines of which  $M$  are idle. Let us assume that there is one center of computation. If it takes an average time  $\delta_1(A)$  which is a function of the average network traffic for a recruitment message to reach another machine on the network,  $\delta_2(A) + L_1$  time for a message to be returned, and finally  $\delta_3(A) + L_2$ , the total time for uploading of a computation from a central server is given by

$$\tau_u = D(A) + L \quad (3)$$

where

$$D(A) = \delta_1(A) + \delta_2(A) + \delta_3(A) \quad (4)$$

$$L = L_1 + L_2 \quad (5)$$

$L_1$  and  $L_2$  are *lag times*, and represent the time it takes for a given processor to process a given message.

We may assume that the computations themselves may be partially completed on a given machine and completed elsewhere if the computation cannot be completed in a specific amount of time. If this is the case, then if each processor may be limited to a number of computations equal to

$$N_c = \frac{Max}{T_c} \quad (6)$$

where  $T_c$  is the average time per cycle on a given machine, then the computation will be completed on a number of machines equal to

$$C_t = Max \sum_{i=1}^{N_{c_m}} \frac{1}{T_c} + N_{m_c} (D(A) + L) \quad (7)$$

where  $N_{m_c}$  refers to the number of machines on which this computation was placed before completion. Note that the units of  $T_c$  are *computations time*<sup>-1</sup> and that those of *Max* are *time*. If we have many computations

of this type, spread across the entire network, the total number of computations will be given by

$$Total = KC_t \approx KN \left\langle \frac{1}{T_c} \right\rangle ((Max) + (D(A) + L)) \quad (8)$$

where  $K(Max + (D(A) + L))$  refers to the total time of the computation and

$$\left\langle \frac{1}{T_c} \right\rangle = \frac{1}{N} \sum_{i=1}^N \frac{1}{T_{c_i}} \quad (9)$$

Moreover, the contribution of any given machine, in terms of the number of computations will be given by

$$C_i = \frac{K(Max + (D(A) + L))}{T_{c_i}} \quad (10)$$

which is the optimal use of the processor. If  $Max \gg D(A) + L$  then

$$C_i \approx \frac{K(Max)}{T_{c_i}} \quad (11)$$

which is equal to the number of computations the processor could possibly put out in  $K(Max)$  time, if it was running continually. Thus, this simple and minimal method can optimally utilize the processors in the network.

### 2.2.3 Equipartitioning

We turn our attention to the equipartitioning of the network, given the existence of two or more processors serving as centers of computation. We start by assuming that the processors are not correlated either in their timing or in the order of their request sweeps. We also assume that the timing of the individual processors is uncorrelated with the individual temporary central processors. Finally, we assume that there is no preferred processor; that is, any temporary central processor is equally likely to be granted the use of any other processor.

If the individual central machines are uncorrelated in timing either with one another or with other processors on the network, then we may assume that any given idle machine is equally likely to be initially contacted by a given temporary central server. Since the system is equally likely to provide responses to either machine, this means that the initial message to the central processor is equally likely to be sent to either machine. Thus, the probability that central processor  $i$  of  $L$  central processors will be the first to receive a response is

$$p_r = \frac{1}{L}. \quad (12)$$

Once the response has been received, the central processor must send a response to the available processor. This

takes place in a time  $L_{2_i}$ , which is the lag time of the  $i$ th processor. If this message is the first to make it back to the available processor, the computation will be carried out. Otherwise, the available processor will no longer be available. This will be the case if the available processor does not receive a message from another central processor in the time  $L_{2_i} - L_{2_j}$  for any  $L_{2_j} < L_{2_i}$ . This occurs with a probability

$$p = \prod_{L_{2_j} < L_{2_i}} \left( 1 - \frac{L_{2_i} - L_{2_j}}{\gamma_j} \right) \quad (13)$$

Thus, the available processor will carry out computations from the  $i$ th central processor with probability

$$p_i = \frac{1}{L} \prod_{L_{2_j} < L_{2_i}} \left( 1 - \frac{L_{2_i} - L_{2_j}}{\gamma_j} \right) \quad (14)$$

Of course if all the processors are of equal speed, this will become

$$p_i = \frac{1}{L} \quad (15)$$

meaning that the processor will be used exactly equally by all central processors if the processors have identical lag times. Of course this is not always the case, and it means that those processors with smaller lag times will have smaller proportions of the swarm.

In the extreme case of two processors, the difference in lag time may be measured by measuring the average number of processors in use by different central processors. In this case,

$$p_1 = \frac{1}{2} \left( 1 - \frac{L_{2_1} - L_{2_2}}{\gamma_2} \right) \quad (16)$$

if machine 1 is the slower machine. Rearranging, we obtain

$$L_{2_1} - L_{2_2} = \gamma_2 (1 - 2p_1) \quad (17)$$

If the machines are different classes, then the partitions would be expected to have the more complicated form

$$p_i = \frac{s_i}{\sum_k s_k} \prod_{L_{2_j} < L_{2_i}} \left( 1 - \frac{L_{2_i} - L_{2_j}}{\gamma_j} \right) \quad (18)$$

In the case of two differing computers, this would have the form

$$p_1 = \frac{s_1}{s_1 + s_2} \left( 1 - \frac{L_{2_1} - L_{2_2}}{\gamma_2} \right) \quad (19)$$

### 2.2.4 Security

Security is a tricky issue. Earlier, we noted that a well-designed swarm-based system is secure both from unwanted damage to the processor providing processing

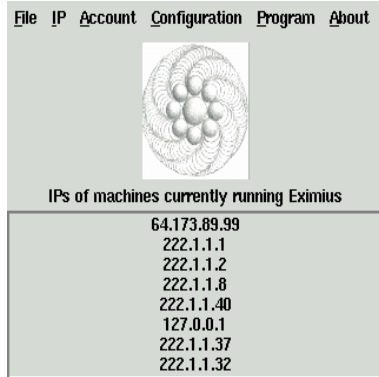


Figure 1: The Eximius front end.

services, and from unwanted eavesdropping on calculations going on. Moreover, data corruption is to be avoided; processors are trusted to carry out the computations correctly. Design implementations which require this to be the case must be explored.

### 3 Eximius Processor

The eximius processor is a software package some 681 kB in size. The processor accomplishes a number of tasks as part of its design, aimed at maintaining an accurate current accounting of other processors running on the processor’s network, maintaining local processes initiated by local users, and managing cooperative computational efforts with remote computational centers. The processor is powered by generalized swarm engineering protocols which allow it to do a number of things including respond to changing network connectivity or processor availability, maintain an accurate list of active processors, recover from processor crashes during active programs, and carry out multiprocessed program completion on local processors.

In this section, we describe some of the processor-specific protocols that make the eximius processor function, both as a member of a swarm and as an individual center of computation. We begin with the basic design of the processor, describing its functionality. We then describe those protocols based on swarm engineering which solve certain problems in a decentralized and interesting way.

#### 3.1 The basic eximius processor

The eximius software package is built around a simple virtual processor capable of accepting and completing a number of computational commands from outside agents which may be local or global computers. The commands are packaged in much the same way that an object in object oriented programming (OOP) might be created: the

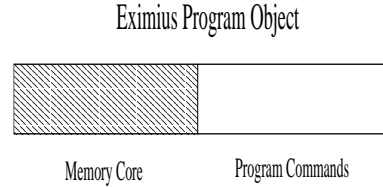


Figure 2: The structure of an eximius object is depicted here. It consists of two distinct parts: the data core and the command set.

object consists of a set of data and code which handles the data. The processor forms the inner most layer of the eximius processor, with other layers built to communicate with this layer and other remote processors.

Eximius objects (EOs) have two main parts. The first part is the *data core*, while the second part is the *command set*. The data core consists of all data, in whatever format is required. The second consists of processing commands. A generic EO is depicted in Figure 3.2.

The eximius object is processed in the central processor. All completed results are saved in the data core, which is returned to the calling program for further processing.

A program utilizing the eximius processor need only produce a number of EOs and upload these EOs to the processor. When the processor has completed these processes, it returns the data to its original location in the program’s memory, informing the program of the return of the data by setting flags set aside to indicate changes in the given variables. If the processor is occupied, and one or more processes is queued for processing, the eximius processor will recruit the assistance of known active processors running on other machines.

In order to be able to recruit computation, each eximius package must be able to upload excess EOs to available processors. In order to do this, each package maintains a list of IP addresses for other machines known to be running the eximius processor. This list is updated from a list of known IP addresses assembled over the time that the processor has been installed on a given machine. Each time a processor is contacted by another known processor, the IP is recorded, and the new processor may be contacted the next time a particular processor is initiated.

Each Eximius processor is configurable. The configuration includes the maximal number of cycles that a processor will compute before returning the unfinished process to the original center of computation. As in Section 2, this is limited by a predetermined amount of computational time, which may be determined by the user, but in general is determined automatically, on installation.

### 3.2 Swarm extensions to the processor

One of the important design paradigms is that the processor be able to become completely integrated into the group of other processors, without the need for direct user interaction. Moreover this should happen throughout the swarm, with gradual adaptation of the swarm to the availability or lack thereof of computation. This is accomplished in the Eximius software package by the maintenance of a listing of available processors and a peer-to-peer based process by which new or newly active processors may become known to other processors.

Each processor maintains a list of known processors which have become known at some time in the past to the processor. This list is updated each time a new processor makes itself known. This list, the known processors list, constitutes the set of all machines known to the processor, active or not. Independently, the Eximius processor maintains a list of active processors. These are processors known to be active at the last interaction between Eximius processors. This list defines the set of processors contacted if new processing power is required for any given reason.

Occasionally, a particular machine or network fails. In such a case, the central processor for a given computation will requeue any processes on any affected machines, effectively terminating the collaboration. Moreover, the machine or machines affected will be removed from the active processor list, eliminating them from the local image of the swarm. In this way, the individual machine can make sure that remote machine failures do not affect the overall computation under way.

When new machines become available once again, the machines initially send signals to all machines in the known processors list. Those processors currently running the Eximius software are moved to the known active processors list. This re-integrates the machine into its known swarm, though it will ignore those machines currently not running the Eximius software, if up at all. For processors which have not connected with any members of the swarm, this list can be short, containing only a small set of machines, which may be default machines.

The swarm protocols center around the gradual increase in the number of machines known to an individual element of the swarm. Periodically, each Eximius processor chooses two distinct members of its known active list and sends a hello message to one of them in which the sending machine masquerades as the second machine from the list. This evokes a response from the recipient machine which is sent to the other machine. In the case that the second machine from the list is unknown to the first machine from the list, and vice versa, the two ma-

chines have learned of one-another's existences, and so the perceived swarm from the point of view of each machine has increased in size. This process also has the effect of allowing the original machine to check on the status of a processor perceived to be active. Inactive processors may be identified at this point, avoiding their detection during a computation. For clarity, we identify this process as *rebroadcasting*.

We may calculate the expected time it might take for a particular processor to become known to the entire swarm. Assume, first that there are  $N$  processors where  $N \gg 1$ . Then, the addition of a single processor only known to a single other processor which is part of the swarm will produce a probability that a signal will be sent to another processor about this particular processor of

$$p \approx \frac{N}{N+1} \frac{1}{N} = \frac{1}{N+1} \quad (20)$$

and the rate at which the particular signal will be sent to other processors will be

$$r = \frac{1}{\delta_r} \frac{1}{N+1} \quad (21)$$

where  $\delta_r$  is the rebroadcasting rate of the software. We may then rewrite this as

$$\frac{dS}{dt} = S \frac{1}{\delta_r} \frac{1}{N+1} \quad (22)$$

which gives the number of knowledgeable swarm elements  $S$  by

$$S(t) = e^{\frac{t}{\delta_r(N+1)}}. \quad (23)$$

This gives the time to completely become known to the swarm as

$$\delta_r (N+1) \ln(N) = t_S. \quad (24)$$

This means that the propagation time of new processor information across the entire swarm grows log-linearly with the size of the cluster.

## 4 The use of the Eximius processor

We investigate the use of the Eximius processor by measuring its effectiveness in completing a computation involving 3000 individual computations, each of which has up to  $10^6$  individual computations. Thus, the total number of computations is  $3 \times 10^9$ . This computation is an extremely long calculation generating a Mandelbrot set, which has a maximum depth of  $5 \times 10^5$  levels, and with each individual computation comprising the set of calculations required to set the color of a specific pixel in the image. Though the generation of a Mandelbrot set need not have this depth, the purpose of the computation is to explore the Eximius processor.

All calculations are run on a small cluster of between ten and thirty PC computers ranging from Pentium I class machines running at 133 MHz to AMD Athlon-based PC computers running 1.3 GHz. As the software is developed in Linux, all processors are run under the Linux operating system, and report the amount of computation completed on remote machines. In order to test the linearity of the system, we run the calculation on a remote machine on which the local processor has been deactivated. We run the calculation using different numbers of processors, and report on the time of computation and the amount of computation completed on each machine.

In order to measure the partitioning of the network, the computation (a single program) was run on one machine, and automatically recruited the efforts of other machines. The program recorded the computations carried out on other computers in order to determine the partitioning capability of the system. We report the total number of computations done on computers, and their relative speeds. We do not report the partitions running on machines in use by computers other than those that are comparable and in dedicated use, as their speeds are not comparable to the number of computations carried out. We also report the computation time of the full computation as we vary the computational power of the network by bringing various parts of the network on between various computations.

Computer	Speed	Comp.	$\frac{speed}{comp.}$
8	1300 MHz	8360	0.155
21	501 MHz	1408	0.355
24	451 MHz	1320	0.342
26	1300 MHz	8292	0.156
32	1300 MHz	8470	0.153
33	1300 MHz	8360	0.156
36	1300 MHz	8448	0.154
37	1300 MHz	8128	0.160
38	1300 MHz	8294	0.157
39	1300 MHz	8360	0.156
40	1300 MHz	8426	0.154

Table 4.1: This table gives the partitioning of the network in the presence of a heterogenous set of computers. Notably, the partitioning of the faster machines and slower machines is very consistent, though the slower machines seem to be partitioned very differently from the faster machines.

In Table 4.1, we report on the partitioning of the network. The last column on the right gives the relative speed vs. computation throughout the overall computation. Notably, the ratios are nearly identical within classes of machines, but significantly different between classes. This is not as predicted, except if the times are

different. In this case, we may calculate the approximate value of  $\frac{L_1-L_2}{\gamma}$  as 0.48. This means that the difference in lag times is significant on the different class machines, and is an important part of future eximius system design.

## 5 Discussion and concluding remarks

We have presented in this paper a software package designed for non-centralized computational resource sharing. This software package’s main design paradigms, enumerated in Section 2 have been accomplished via methods inspired by the Swarm Engineering paradigm of design. In this method, a single simple condition is generated which, when satisfied, guarantees that the final global result is accomplished. In our case, we wanted to produce a system which allowed the sharing of computational resources while allowing unfettered quick access to the swarm, and which produced an efficient use of the available computational resources. This was accomplished by using individual computational agents which employ simple behavioral paradigms whose properties may be shown to lead to the overall global behaviors. This behavior was verified experimentally using a long ( $3 \times 10^9$  computations) computation on a small number of machines in a local computational cluster.

One part of the design paradigm we have not touched on is the security of the overall computation. This is perhaps one of the biggest concerns from the point of view of the individual computer owner and individual user of Eximius. While the protocols utilized in the current version of Eximius do not currently attempt to provide security to the user, the current version of Eximius provides excellent security for the individual computer owner. As all computations are objects with specialized functionality and memory, the EO which is uploaded can be very carefully controlled. Error trapping routines are in place which guarantee that errors caused by memory overruns or faults caused by improperly written code may be caught without crashing the processor or causing exploitable faults. Moreover, since the computations are provided via EOs, the processor need not have access to any data stored outside of the EO’s memory core. This effectively removes any possibility of the uploading of a “Trojan Horse” EO.

On the other hand, the possibility of some kind of terrorism, espionage, or corruption in the data sent out and received is a very different and more difficult to correct problem, which has received attention before [2]. This stems from the potential creation of “rogue processors” which produce bad data, bad transmission lines which corrupt data coming back, or the collection of data sent out to a particular processor for completion. Of these, the last is the most unlikely, as the computations are not

necessarily put in any particular order, and are sent to a large number of uncorrelated computers. The others may be solved by using voting schemes or believability measures, though these either require an increase in computation or some kind of recording of the interactions with processors and their ability to perform predetermined calculations correctly.

The use of swarm paradigms to solve these problems in the same way as the other problems have been solved is a topic of future research. These problems must be solved in order to make the Eximius processor or related processors viable solutions for secure computation on a global distributed supercomputer. However, the power of the use of simple communication protocols in generating the high level results described above would seem to indicate that swarm engineering-based methods may lead to the development of a secure computational environment which simultaneously handles the need for efficient use of resources.

The great power of the Eximius processor stems from the ability of the user to initiate any computation on any computer. This removes the need for central servers and the sale of computation services either provided using large computers or distributed networks of computers. Moreover, software packages utilizing the Eximius protocol may immediately take advantage of the ever-improving computational environment, offloading computation to the network while handling graphical user interface on local machines. The potential for huge computations, entertainment in the form of interactive video games, and a plethora of other uses is virtually unending. Moreover, the potential for the average computer user to make use of the worldwide computational fabric is significantly enhanced by Eximius and Eximius-like processors if software packages which might take advantage of the network can be developed. Such a developing decentralized protocol would seem to be necessary to allow the development of next-generation computers whose main computation is relegated to the network, and whose GUI may be built in a hand-held unit that could fit in the pocket. Such systems have the potential to make it possible for supercomputer-class computational resources to be carried around as easily as a cell phone, making huge computational problems capable of being carried out from virtually any location, on any machine, at any time.

## 6 Acknowledgements

The Eximius processor system was produced primarily by Sanza Kazadi and Daniel Lee at the Jisan Research Institute. Other people who assisted in the development of the software package were Willie Chen, Elliot

Acevedo, Henry Lin, Andy Hsieh, Peter Hung, and Paul Hung. The software will be made available at the website of the Jisan Research Institute sometime during the summer of 2003.

## References

- [1] S. Kazadi. **Swarm Engineering**. California Institute of Technology PhD Thesis, 2000.
- [2] D. Arapov et. al. *A programming Environment for Heterogenous Distributed Memory Machines*. **Proceedings of 6th Heterogenous Computing Workshop (HCW'97)**, IEEE Computer Society, Geneva, Switzerland, 32-45, April 1997.
- [3] A. Natrajan et. al. *Grids: Harnessing Geographically-Separated Resources in a Multi-Organizational Context*. **High Performance Computing Systems**, June 2001, (Keynote Speech).
- [4] A. Natrajan et. al. *The Legion Grid Portal*. **Concurrency and Computation: Practice and Experience**, 14.
- [5] I. Foster, C. Kesselman, and S. Tuecke. *The Anatomy of the Grid. Enabling Scalable Virtual Organizations*. **International Journal of Supercomputer Applications**, 2001.