Balart, J. [et al.]. Experiences parallelizing a web server with OpenMP. "Lecture notes in computer science", 2008, vol. 4315, p. 191-202. The final authenticated version is available online at https://doi.org/10.1007/978-3-540-68555-5_16

Experiences parallelizing a Web Server with OpenMP

Jairo Balart, Alejandro Duran, Marc Gonzàlez,
Xavier Martorell, Eduard Ayguadé, Jesús Labarta

CEPBA-IBM Research Institute
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain.
{jbalart,aduran,marc,xavim,eduard,jesus}@ac.upc.edu

Abstract. Multi–threaded web servers are typically parallelized by hand using the pthreads library. OpenMP has rarely been used to parallelize such kind of applications, although we foresee that it can be a great tool for network servers developers. In this paper we compare how easy is to parallelize the Boa web server using OpenMP, compared to a pthreads parallelization, and the performance achieved. We present the results of a parallelization based on OpenMP 2.0, the dynamic sections model and pthreads.

1 Introduction & Motivation

OpenMP [1] has successfully been used to parallelize a great number of applications in the scientific domain. Extensive work has been done to fulfill the needs of scientific applications in shared memory environments. The parallelism that appears in numerical applications has significantly influenced the definition of the OpenMP API. Most work distribution schemes are specifically designed to support the main source of parallelism of scientific applications: parallel loops. For synchronization mechanisms programmers can use barrier synchronizations, mutual exclusion and atomic synchronizations.

But scientific applications are not the only niche where parallelism can be used to increase the performance of applications. In fact, with new generations of architectures containing multiple cores, parallelism will be exploited in applications with characteristics and needs dramatically different from those of the scientific world. We believe the OpenMP community should start studying these characteristics. This study will either determine if the current OpenMP API suites these new applications or whether it needs changes to support them efficiently.

We have studied the feasibility of using OpenMP to parallelize a web server to start exploring the characteristics that will be found in these new applications. Web servers are inherently parallel applications as the different requests are unrelated and free of dependences between them. Thus, the requests can

be handled in parallel. Web servers have been traditionally parallelized with *threading* techniques (mainly *pthreads*). Our objective is to specify useful extensions if the OpenMP API does not easily support the parallelism found in web servers.

We have selected the Boa [2] web server as the platform to be parallelized. Different parallel strategies have been developed. An OpenMP version allowed us to evaluate the programming effort and the resulting performance using the current OpenMP standard. Also, we developed a version that uses the proposed *dynamic sections* [3] constructions to check their usefulness. Finally, a manual *pthread* based version was developed to do a comprehensive comparison. All three versions were evaluated against the original version which is single threaded.

The structure of the paper follows: section 2 describes the contributions of this paper to the state of the art. Section 3 overviews the structure of the Boa web server. In section 4 we describe our parallelized versions of Boa. Section 5 describes the experiments performed and the results obtained. Section 6 discusses our experiences with the different parallel versions. And finally, in section 7 we present the conclusions of this work and we describe some lines for future study.

2 Related Work

Several authors have compared OpenMP versus pthreads. These comparisons have been in scientific applications or comparisons between basic language constructions. Kuhn et al. compared the primitives provided by both models concluding that OpenMP was easier to use but they also pointed out some problems with irregular applications [4]. Lee and Downar compared both languages for a nuclear reactor transient code [5]. They obtained similar performance with both OpenMP and pthreads but OpenMP was easier to use. Breshears and Luong compared both models in the context of a Coastal Ocean Circulation Model [6]. Their conclusion was that OpenMP was easier to use yielding the same performance than pthreads. Dedu et al. compared both models with some algorithms from the artificial intelligence field [7]. They found that although for regular applications OpenMP was easier to use and obtained the same performance than pthreads for irregular applications OpenMP was difficult to use.

Some other works have tried to extend OpenMP to be able to cope with irregular applications. Asenjo et al. explored some techniques to deal with pointers and traversal of structures [8]. Shah et al. introduced the *workqueueing model* [9]. This proposal extends the OpenMP programming model with an alternative work distribution scheme based on the definition of *queues of work* from where the executing threads extract work. The extension targets algorithms traversing memory and linked data structures. The proposal has been successful but introduces dramatic changes in the OpenMP execution model. We previously proposed to use *dynamic sections* [3] to minimize the changes of the

workqueueing model, in which *dynamic sections* are based. We further extend the semantics of the *dynamic sections* in this work.

Different works have evaluated the usefulness of threaded web servers. For example, Roper et al. compared different web servers concluding that multi–threaded servers can outperform traditional event driven servers [10] which are not threaded. Jeong et al. showed in their work [11] that the use of multiple CPUs with dynamic content increases the throughput obtained and reduces the response time. Using multiple CPUs with static content does not increase throughput but response time still decreases. The benefits of using multiple CPUs in SSL enabled web servers were shown by Guitart et al.[12]. Other successful multi–threaded servers include Apache [13], SEDA [14] and Flash [15]. All these works used a system thread packages, mainly pthreads. This work, instead, explores the suitability of OpenMP as a language for the development of a parallel web server.

3 The Boa web server

The Boa web server architecture is a single threaded event–driver HTTP server architecture. This kind of architecture, unlike traditional web servers, does not fork for each incoming connection. Instead, Boa comes with an integrated task scheduler that handles multiple requests concurrently but not in parallel. Boa multiplexes all ongoing requests, trying to maximize the throughput and minimize the response time. The scheduler uses two request queues: the *ready* queue keeps those requests available for further processing. The *blocked* queue keeps those requests waiting for any data dependence to be satisfied. Iteratively, the server traverses the *ready* queue and further processes each request. The server uses a round–robin technique to avoid large requests starving other *ready* requests. Boa logically divides each request in smaller chunks of work and each time a request is processed a single chunk is consumed. Figure 1 shows a simplified code that processes the requests from the *ready* queue. After a request is processed

1. It is kept in the *ready* queue because it should be further processed (i.e. it has more chunks and all data are available).
2. It is moved to the *blocked* queue because the server detected an unsatisfied dependence (e.g. it needs to read data from a socket).
3. It is freed because the last chunk of the request was consumed.

The server traverses the *blocked* queue and for each request it checks if the request dependences are satisfied using the information it collects with the *select* system call. When a request has no further dependences it is moved again to the *ready* queue.

Figure 2 shows the structure of the main loop of the server. This loop is infinite and in each iteration the server

1. processes any pending signal.

```
for each request in the ready queue
{
        update_time;
        result = process_step(request);
        accept new requests (if any);
        if ( result == BLOCK )              block(request);
        else if ( result == FINISHED )      free(request);
        else keep it in the queue
}
```

Fig. 1. Request processing loop pseudo–code

2. traverses the *blocked* queue to check the dependences of blocked requests.
3. establishes pending new connections using the *accept* system call.
4. traverses the *ready* queue to process more chunks of the unblocked requests.
5. calls *select* to obtain information about new connections and the status of incoming and outgoing data.

```
while (1)
{
        process signals (if any)
        move requests from blocked to ready using select result
        accept new connections (if any)
        process requests in the ready queue
        select system call
}
```

Fig. 2. Main loop pseudo–code

Boa tries to reduce the number of issued system calls by *mmaping* local files into the server memory. A cache of *open files* (i.e active maps), which it is checked each time a new file is requested, avoids *mmaping* twice the same file.

The server performs all input/output with non-blocking system calls to ensure that no single request blocks the processing of others that are ready.

4 Parallelizing Boa

The main source of parallelism in the Boa web server is the possibility of overlapping the computations related to different requests. As explained in section 3 the requests are placed in the *ready* and *blocked* queues and the server iteratively traverses these queues. Parallel processing is possible by processing each element of the queues in parallel. The server can also do different tasks in parallel (e.g. accepting new requests and processing new requests). All the versions described exploit these sources of parallelism.

Different points in the server require serialization. First, modification of global variables (e.g. the number of active connections). Second, manipulations

of the *ready* and *blocked* queues. Third, acceptance of new connections. Fourth, access to the cache of open files. And last, write access to the server log files to avoid mixing the output of different threads.

The original version uses a lot of *static* variables insides functions. These variables were converted to extra parameters of the function they were in. Another possible option was to use per thread variables (e.g. *threadprivate* in OpenMP).

We have developed three parallel versions: a *pthreads* version, a pure OpenMP version and a version using *dynamic sections* which are non–standard.

4.1 Pthreads parallel version

The *pthread* parallel version exploits the possibility of processing different requests in parallel, as they are unrelated. The parallelization uses a producer–consumer approach. One thread executes all the tasks in the main Boa loop, described in Figure 2, except requests processing. This thread is the producer of new *ready* requests. The remaining threads consume the *ready* requests and process them as explained in section 3. Figure 3 shows the code the consumer threads execute. The *pthread* version uses the same round–robin mechanism of the serial version. So, each time a thread extracts a request a single chunk is consumed and the request may be queued again to the *ready* queue.

This version uses different mutex locks to protect accesses to the *ready* queue, accesses to the *blocked* queue, accesses to global variables, and writing to the server log files. The cache of open files is also protected by a mutex lock per entry, to maximize concurrency, and a global mutex lock for global cache variables.

```
1  for ( ; ; ) {
2    while( not pending requests );
3    pthread_mutex_lock(&ready_lock);
4    if ( pending requests ) {
5      req = dequeue(request_ready);
6    }
7    pthread_mutex_unlock(&ready_lock);
8    if ( req )
9      process_request(req);
10 }
```

Fig. 3. Code for thread consumers in the *pthread* Boa version.

4.2 OpenMP standard parallelization

For our OpenMP parallelization we targeted the request processing loop Figure 1. We wanted to distribute all the requests in the *ready* queue among the available threads by using a workshare. The number of requests in the queue can vary during its traversal (e.g. if a request is free). Because in OpenMP all

threads must see the same iterations we splitted the loop in two new loops: one does not remove requests the queue while the other does. In the first new loop, the server processes each request in the *ready* queue. The result of each processing is stored in a new field of the request structure. This loop was parallelized using a *parallel* construct and we used a *single* workshare to distribute the different iterations. Using the result from the first loop the server modifies the queues in the second loop which must be done single-threaded. Figure 4 shows the OpenMP parallelization of the new loops. When new requests are accepted they are added to the beginning of the queue. Before the server accepts any request all threads grab the head of the queue. This guarantees they will traverse the same elements.

```
1  #pragma omp parallel
2  {
3
4          get head of ready queue
5          #pragma omp barrier
6          for each request in the ready queue
7          {
8          #pragma omp master
9                  update time;
10         #pragma omp single nowait
11                 request.result = process step(request);
12         #pragma omp master
13                 accept new requests (if any);
14         }
15 }
16
17 for each request in the ready queue
18 {
19         if ( request.result == BLOCK )                block(request);
20         else if ( request.result == FINISHED )        free(request);
21         else keep it in the queue
22 }
```

Fig. 4. OpenMP request processing loop pseudo–code

Access to the cache of open files was protected with a critical section for the global cache variables and an OpenMP lock per cache entry. We used another critical construction to guarantee correct access to the log files. Several critical sections protect access to Boa global variables.

4.3 Dynamic sections parallelization

The previous presented parallelizations were in previous section were mainly possible because the serial version had already embedded a complex code that dealt with request blocking and their scheduling (i.e. the ready queue). Our question now is: could OpenMP make this work easy to the programmer?

The available parallelism can be seen as a collection of tasks. We have used the *dynamic sections* proposed extension to express this parallelism easily. Under

this model a single thread is in charge of performing the serial work (accepting requests, extracting them from the blocked queue, ...) while the remaining threads execute the parallel tasks that are created (i.e. a dynamic section).

In this version, we have removed part of the integrated schedule: the ready queue has been removed. Instead of queueing requests in the *ready* queue now the threads create new dynamic sections. A new dynamic section is created

- When a new request is accepted, a new section is created for the first chunk of the request.
- When a request has completed a chunk, if it was not the last, a new section is created for the next chunk. This dynamic section is created inside another. We have extended the original model to allow nesting of SECTION constructs.
- When a request is unblocked, because their dependences are fulfilled, a new section is created for the next chunk.

```
#pragma omp parallel
#pragma omp sections dynamic
while (1)
{
        process signals (if any)
        foreach request from the blocked queue {
                if ( request dependences are met ) {
                        extract from the blocked queue
#pragma omp section captureprivate(request)
                        serve_request(request)
                }
        }
        if ( new connection ) {
                accept it
#pragma omp section captureprivate(new connection)
                server_request (new connection)
        }
        select system call
}
```

Fig. 5. Main loop pseudo–code with *dynamic sections*

Figure 5 shows our parallelization of the code of the main loop using the *dynamic sections*.

As in the previous version, several critical constructions protect accesses to the open files cache, access to the global variables and access to the server log files.

5 Evaluation

5.1 Environment

For our experiments we used a 4-way Intel Xeon at 1.4GHz with 2GB of RAM to run the web server and a 2-way Intel Xeon at 2.4 GHz with 2GB of RAM to

run the benchmark client. All the machines were running a 2.6 Linux kernel. The network that connected the machines was a switched Gigabit network.

5.2 Workload generator

We used Httperf[16] to generate the different workloads for the experiments. This tool allows the creation of a continuous flow of HTTP requests to the server machine. The tool accepts as one of its parameters the number of clients per second. For each client, it opens a session with the server through a persistent HTTP connection. Then a series of requests are issued by the client, some of them pipelined, some spaced by a *think time*. Another parameter of Httperf is the sessions database from where clients get the requests they ask for and the think times to wait. We have used a database extracted from the Surge[17] workload generator.

The scenario produced by Surge is a static content workload characterized by short session lengths and low computational costs for each request serviced. The Surge distribution is based on a model developed from the observation of real web server logs.

5.3 Experiments

We evaluated all different versions of the Boa web server using the Surge workload with different loads of clients. These load configurations ranged from a low load of clients (10 per second) to heavy load of clients (800 clients per second). In the following plots, we labeled the different Boa versions as follows:

- *original boa* refers to the unmodified single–thread Boa server.
- *boa-pthreads* refers to the parallel version that uses *pthreads*.
- *boa-omp* refers to the parallel version that uses standard OpenMP constructions.
- *boa-dsections* refers to the parallel version that uses *dynamic sections*.

All parallel versions were run with 2 and 4 threads.

Figure 6 shows for each load of clients the throughput obtained by each Boa version. All versions, except the *boa-omp* version, obtained a similar throughput up to a workload of 700 clients per second. The *boa-omp* version was outperformed because the server did not run as much time in parallel as the other versions do. Doubling the number of threads in the parallel versions did not result in a noticeable increase of throughput because Surge is not CPU–intensive workload as it works with static content. With 700 clients per second the limit of the Gigabit network was reached and all versions throughput deteriorated as they were saturated. Saturation happens earlier using more threads because contention in shared resources have a greater impact. The Boa server uses a mechanism that minimizes the effect of saturation limiting the number of active connections. We disabled this mechanism on purpose so we could find the point at which each parallel version saturates.

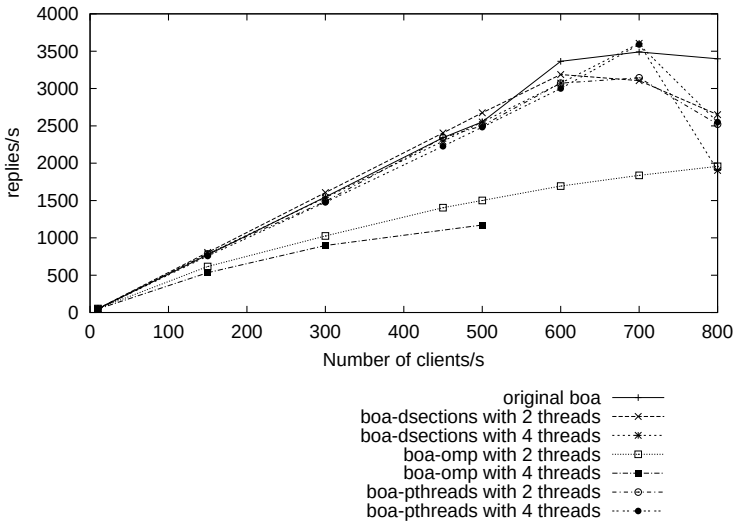

Fig. 6. Throughput (replies per second)

Figure 7 shows the average response time for each load of clients and each Boa version. All parallel versions achieved a lower response time than the original Boa version. With a load of 800 clients per second response time was reduced as much as by three. Doubling the number of threads in *boa-dsections* and *boa-pthreads* was useful reducing the response time. With four threads *boa-dsections* and *boa-pthreads* behaved so closely that their lines are overlapped. With two threads they behaved similarly except with a load of 700 clients per second where *boa-pthreads* response time was lower. The average response time for the *boa-omp* version was very low, near to zero. This result is misleading because the server was rejecting more than 75% of the requests.

6 Comparison

In this section we compare the programming effort required by the three parallel versions: *pthreads*, OpenMP and *dynamic sections*.

One of the most consuming tasks in parallelizing all the versions was removing the static variables in local functions. Due to the large amount of *static* variables that may appear in serial C codes compilers could provide an option that transformed any variable with *static* storage to a variable with *thread private* storage. This option would reduce time spent in parallelizing large C codes.

Another effort, common in all versions, was protecting shared data with critical sections and locks. This work in was quicker done with OpenMP than with *pthreads* as you only need to add the appropriate directive instead of having to declare a mutex variable and using *lock* and *unlock* calls. Nevertheless, when you need a critical section for each element of a structure (e.g. the open

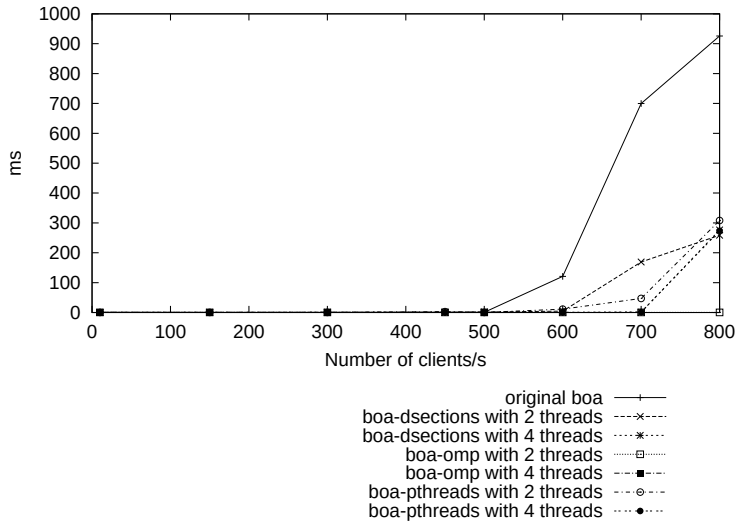

Fig. 7. Response time (ms.)

files cache) for improving performance, it is as difficult in OpenMP as it is in *pthreads* because you need to use omp_locks. We think a possible way to solve this problem would be allowing dynamically named critical sections (e.g. cache_lock[i]). With these kind of critical sections the same code protecting a single entry of an complex structure would work for all the entries while each one will still have its own lock.

The *pthread* version required several modifications to the original source code. Code for the consumer threads was developed. And, although it was not very difficult because there was only one type of task to consume (i.e. requests) it required some expertise to handle access to the queue correctly and efficiently. Some other changes were required to the code of the producer thread including: creating the consumers, initializing the locks, and removing the request processing loop. Also, mutex locks were added to protect the access to the *ready* and *blocked* queues. The overall effort was moderate.

The OpenMP version did not require as many changes in the source code compare as the *pthread* version. But, it needed a great deal of attention to maintain the correctness of the *single* workshare (i.e. that all threads executed all the iterations). This restriction also caused the reduction in performance. In the other versions, the threads could execute a task as soon as it was ready, or even run in parallel request processing and accepting new connections. In the OpenMP version all threads must wait until all the requests, that were available when the traversal started, are processed even if there are new requests to process. This suggests that current workshares are not well suited for handling irregular parallelism.

But both the *pthread* version and the OpenMP version were easier to develop because the original version had integrated a complex code that enabled request concurrency. Otherwise, the programming effort would have been greater. On the other hand, the *dynamic sections* version was simpler as the programmer did not need to code the management of the *ready* tasks. Even from a simpler version of the serial version the programming effort with *dynamic sections* would have been minor.

Dynamic sections also allow to easily mix different kinds of parallel tasks. While the *pthreads* code would became more and more complex if it had to deal with different kinds of parallel tasks the *dynamic section* complexity would remain constant.

7 Conclusions and Future work

In this paper, we have explored the use of OpenMP to parallelize a web server. We have shown how, adding a few directives, the request processing loop of the web server can be parallelized. But this simple version did not perform efficiently. We used the proposed *dynamic sections* to implement a simpler parallel web server (i.e. without application level task management). Evaluation showed that this version had a performance, in throughput and average response time, as good as the performance obtained by the server developed with *pthreads*. But in the *pthread* version the programmer needed to develop a specific task management for the application whereas the *dynamic sections* version simplified the programming.

In the future, we will apply OpenMP to other web scenarios where studies have pointed out that there can improvements in throughput by using multiple processors: SSL enabled applications[12] and dynamic content applications [11].

Acknowledgements

Authors will like to thank Vincenç Beltran, David Carrera and the eDragon team [18] for their valuable help and for allowing us to use their resources. This research has been supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01.

References

1. OpenMP Organization. Openmp fortran application interface, v. 2.0. *www.openmp.org*, June 2000.
2. Larry Doolittle and Jon Nelson. Boa webserver site. http://www.boa.org.
3. Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP 2004*, October 2004.

4. Bob Kuhn, Paul Petersen, and Eamonn O'Toole. Openmp versus threading in c/c++. *Concurrency - Practice and Experience*, 12:1165–1176, 2000.
5. D. J. Lee and T. J. Downar. The application of posix threads and openmp to the u.s. nrc neutron kinetics code parcs. In R. Eigenmann and M.J. Voss, editors, *Proceedings of the International Workshop on OpenMP Applications and Tools 2001*, volume 2104 of *Lecture Notes in Computer Science*, pages 69–83, July 2001.
6. C. Breshears and P. Luong. Comparison of openmp and pthreads within a coastal ocean circulation model code. In *Workshop on OpenMP Applications and Tools*, July 2000.
7. Eugen Dedu, Stephane Vialle, and Claude Timsit. Comparison of openmp and classical multi-threading parallelization for regular and irregular algorithms. In *Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD)*, pages 53–60, 2000.
8. R. Asenjo, F. Corbera, E. Gutiérrez, M.A. Navarro, O. Plata, and E.L. Zapata. Optimization techniques for irregular and pointer-based programs. In *Proceedings of the 12th EuroMicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*, pages 11–13, February 2004.
9. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallellism in openmp. In *1st European Workshop on OpenMP*, September 1999.
10. Takashi Ishihara, Aaron W. Keen, Justin T. Maris, Eric Wohlstadter, and Ronald A. Olsson. Cow: A cooperative multithreading web server. In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 991–996, June 2002.
11. J. Jeong, S. Park, and J. Nang. Performance analysis of a multithreaded web server on multiprocessor. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 1885–1889, June 2000.
12. J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. In *19th International Parallel and Distributed Symposium (IPDPS'05)*, April 2005.
13. Apache web server. http://www.apache.org/.
14. M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243, October 2001.
15. V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server, 1999.
16. D. Mosberger and T. Jin. httperf – a tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67, June 1998.
17. P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurament and Modeling of Computer Systems*, pages 151–160, 1998.
18. eDragon Research Group site. http://www.ciri.upc.edu/edragon.