

VISUALIZING OBJECT AND METHOD GRANULARITY FOR PROGRAM PARALLELIZATION

WILLIAM HUI†, STEVE MACDONALD, JONATHAN SCHAEFFER AND DUANE SZAFRON

Department of Computing Science, University of Alberta
{stevem,duane,jonathan}@cs.ualberta.ca,
<http://www.cs.ualberta.ca/~{stevem,duane,jonathan}>

†Current Affiliation: Alcatel; whui@alcatel.com

Keywords: Visualization, Object, Parallel Programming, Profiling

Abstract

There are increasing demands for more computing power. Parallel hardware is now commonplace and could be a cost-effective solution. However, to many developers, parallel programming is not a user-friendly task. At the same time, many programmers are turning to object-oriented techniques. Unfortunately, parallel computing with objects introduces many new problems. Tools are needed to help programmers convert their sequential object-oriented programs into parallel ones. This paper introduces Revy, a platform-neutral, easy-to-use, object/method granularity visualization system that assists parallel programmers in transforming their sequential object-oriented programs into parallel ones. Revy allows users to view and inspect the object communication patterns of their sequential applications. It also serves as a profiling system that helps them identify the high-granularity objects/methods as candidates for parallel execution. This paper describes the requirements, architecture and implementation of Revy, illustrating the ideas with a case study.

Introduction

Networked and multiprocessor workstations provide users with a large amount of computing power by running applications on many processors at once. This hardware is now commonplace and is a cost-effective approach to solving computationally-intensive applications. However, the software advances necessary to exploit this power have lagged behind the hardware advances.

Writing parallel software is often perceived as a complicated endeavor. It is more difficult to design, implement and test parallel software than comparable sequential software. Parallel programming includes issues like communication, synchronization, deadlock, task granularity and concurrent non-deterministic behavior.

In the object-oriented programming paradigm, parallel programs introduce additional problems. Real world objects are generally autonomous entities whose activities are performed concurrently. At first glance,

objects and parallel programs seem to be a perfect match. However, this is not always the case. Problems with inheritance, encapsulation, and reusability are outstanding research issues for concurrent object-oriented language designers since inheritance and synchronization often conflict [1] [2]. This problem is called the inheritance anomaly and requires the re-definition of inherited methods to maintain the integrity of concurrent objects. Most parallel object-based languages therefore either do not support inheritance [3], or do so by compromising the encapsulation [4] or the reusability [5] properties.

Despite these difficulties, the promise of re-use through objects and high performance through parallelism is quite compelling. Therefore, the number of programmers attempting to write parallel object-oriented programs is increasing. This trend has also been strengthened by the growing use of Java, a language with both objects and threads. To overcome the difficulties of writing parallel object-oriented programs, we need tools that help software developers build these systems easily and efficiently. The dream tool for parallel object-oriented programmers is a parallelizing compiler that automatically transforms a sequential object-oriented program into a parallel one by typing something like: `"javac -parallel application.java"`. However, such compilers are still not available. This paper proposes an alternative approach. We introduce a parallelization advisor, which assists programmers in parallelizing object-oriented applications.

To help develop parallel programs, programmers need a tool to help them visualize object communication patterns [6]. This paper discusses Revy, a tool designed to do exactly this. However, visualizing the communication patterns is not enough. A programmer cannot simply treat each object as a separate process or give each object its own thread, so that it can execute its code concurrently with other objects. There are two reasons for this. One reason is the startup overhead for objects. Creating a new process or thread takes extra time. In sequential object-oriented programs, thousands of objects are created, used and destroyed each second. In the parallel domain the

startup costs for all of these objects would be prohibitive. The second problem is communication costs, either over a network or through shared memory on a multi-processor. Of course network messages are very expensive, but even shared memory accesses are expensive compared to memory cache accesses in a sequential processor.

We define method granularity as the ratio of method computation time to communication time as shown in Figure 1. With a network of workstations, remotely executing the method `alpha()` frees up CPU cycles on the local machine, that can be used for other computations. However, this results in communication overheads that include the times to: pack the argument objects into a network message, send the message, receive the message, unpack the argument objects, pack the result object into a reply message, send the reply message, receive the reply message and unpack the result object.

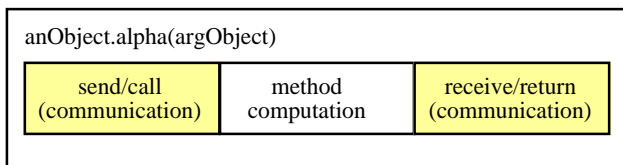


Figure 1. Communication granularity.

On a shared-memory machine, communication overhead consists of the extra time for accessing argument objects from shared memory instead of from local cache, as well as time for writing the result object to shared memory. There are two ways this extra time can be spent. In the first approach, which is analogous to the network approach, the extra time is divided into two blocks, one at the start of the computation and one at the end. In the start block, the argument objects are copied from shared memory to local memory (and then to the cache when they are accessed the first time). The computation accesses the cache for its arguments and stores the result object in local memory. In the end block, the return object is copied from local to shared memory. The communication time is the sum of the times of the start and end blocks. Alternately, the extra time may be spread over the computation by accessing argument objects directly from shared memory as they are needed and writing the result object directly to shared memory as it is computed. In this case, the communication cost is harder to compute. Of course the communication overhead for shared memory access is significantly smaller than for network communication so a computation that has low method granularity on a network will have significantly higher method granularity on a multi-processor.

In either case, programmers must weigh the benefits of freeing up the local processor versus the extra costs of the communication overhead. In object-oriented applications, method granularity plays a more important role than in procedural applications. To increase

reusability and maintainability, object-oriented programs have many short methods instead of a few long procedures. Since the amount of computation time is reduced for each method, this can significantly reduce the communication granularity of computations.

It is not just method granularity that must be analyzed to properly parallelize an object-oriented program. We define the object granularity of an object as the ratio of time that an object is active (executes some code) to the time it takes to create, initialize, delete and re-claim its storage. The cost of creating a parallel object as a process (on a network of workstations) or providing it with its own thread (on a multi-processor) is greater than the cost of creating a similar sequential object as a chunk of memory in the heap. Therefore, parallel objects have smaller object granularities than their sequential counterparts. Again, this granularity problem is worse in object-oriented programs than in procedural ones. In a procedural program, the data structures are often quite large. In object-oriented programs, many simple objects are used instead of a few complex ones to enhance encapsulation and therefore stimulate re-use. For example, a phone number is often declared as an object rather than as a string of 10 digits. The result of this is that most sequential objects do not have sufficient object granularity to be created as parallel objects.

We have argued that method and object granularities play a crucial role in the parallelization of programs in general, and object-oriented programs in particular. Therefore, a tool is necessary to identify the granularities in sequential programs so that they can be efficiently parallelized. In addition to helping developers visualize program communications, such a tool should allow programmers to visualize method and object granularities.

This paper introduces Revy, a platform-neutral, easy-to-use, object/method granularity visualization tool that assists programmers in transforming sequential object-oriented programs into parallel ones. Revy allows users to view and inspect the object communication patterns of their sequential applications. It also profiles programs to identify high-granularity objects/methods as candidates for parallel execution. This paper describes the requirements, architecture and implementation of Revy and illustrates the ideas with a case study.

Revy has three features that distinguish it from existing profiling and program visualization tools:

1. Revy analyses methods on a per-object basis, not just a per-class basis.
2. Revy considers not only the timing statistics, but also the object and method parameter sizes of a user program. Such information is essential because method granularity depends on the time spent transmitting argument data (on networks) or accessing argument data (in shared memory).
3. Revy times object creation and destruction so that object granularity can be computed in addition to method granularity.

The Revy System

Revy is an integrated parallelization advisor, with three research objectives:

1. Explore program visualization techniques to help users understand the object communication patterns,
2. Implement profiling techniques for measuring method and object granularities to identify targets for parallelization, and
3. Investigate parallelization heuristics that suggest the best candidate objects/methods for parallelization.

Traditional profilers compute the time for each individual function to execute and provide statistics like minimum, maximum and average execution time for each function. This allows a programmer to decide whether or not to optimize any particular function by changing the algorithm or by in-lining the code for the existing algorithm. Traditional function execution times correspond to method execution times in an object-oriented application. However, in an object-oriented program it is important to differentiate between times for the same method on different objects. Therefore, some object-oriented profilers report times for methods on a per class basis. For example, a sort method may take longer on instances of the EmployeeList class than on instances of the SalaryList class, since it takes longer to compare two Employees than to compare to Salaries. Therefore, the two methods are analyzed separately for the two classes. However, class-based analysis is not enough. Object-oriented languages support generic List classes whose instances can hold either Employees or Salaries. Therefore, it is necessary to compare the sort method on two different instances of the List class, one that holds Employee objects and one that holds Salary objects.

Revy has three features that distinguish it from existing profiling and program visualization tools. First, Revy analyses methods on a per-object basis, not just a per-class basis. Second, Revy considers not only the timing statistics, but also the object and method parameter sizes of a user program. Such information is essential because method granularity depends on the time spent transmitting argument data (on networks) or accessing argument data (in shared memory). Third, Revy times object creation and destruction so that object granularity can be computed in addition to method granularity.

A data flow diagram of the Revy architecture is shown in Figure 2. Note that rounded rectangles denote major Revy components, regular rectangles denote data and ovals represent non-Revy components.

Revy has three major components:

1. PAS - The Parsing and Annotation Subsystem parses source code and annotates it with instrumentation code. This is the only component of Revy that is language dependent. We currently have two versions of PAS, one for Java and one for C++. Naturally, the language

compiler and run-time libraries (or virtual machine) are also language dependent, but they are not part of Revy.

2. VIS - The Visualization and Interaction Subsystem serves as the interface between the user and the rest of Revy. For example, it displays the entire list of classes and methods in the source code, collects the user's instrumentation directives, displays the call graph and displays the timing statistics.

3. RMS - The Runtime Modeling SubSystem analyses the call graph and the execution traces, computes statistics and provides parallelization hints to the user.

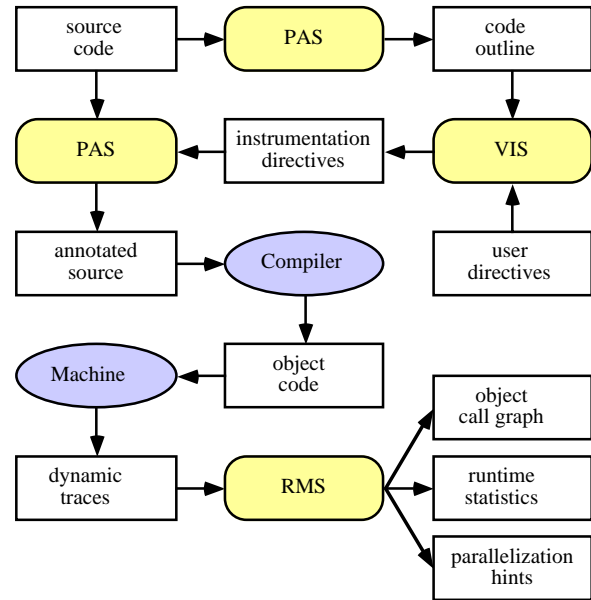


Figure 2. Dataflow diagram for Revy.

Revy parses a user program to identify all of the classes and methods. The program can either be a sequential program or in the case of Java, a parallel program with threads. After parsing the program, Revy presents a list of all classes and methods to the user in a graphical user interface. The user provides instrumentation directives that describe which of the classes and methods are of interest as shown in Figure 3.

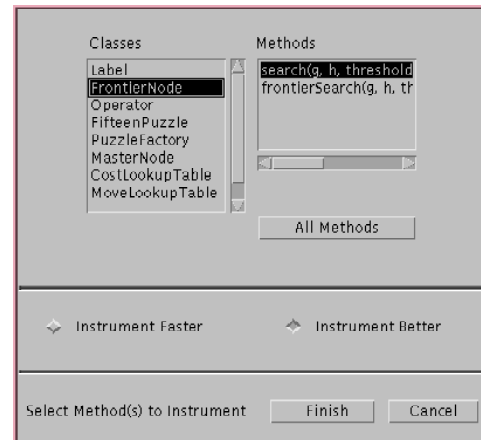


Figure 3. Providing instrumentation directives.

Revy then annotates the source code with instrumentation code. A standard language compiler that is not part of Revy compiles the annotated program. The machine or virtual machine executes the instrumented code to produce trace information which Revy uses to build an object call graph, which is a runtime interaction diagram. Revy then processes the runtime statistics that, together with the object call graph, are visually reported to the user as shown in Figure 7. Finally, Revy provides the user with basic parallelization hints about the program based on the call graph and the statistics.

Since Revy analyses sequential programs, before they are parallelized, actual object and method granularities are not computed. Instead, Revy measures the computation times of methods, the parameter sizes of methods and the active times of objects. Given a particular parallelization architecture, the method communication times and parallel-object creation times can be estimated. These estimates can be used to approximate the method and object granularities. For example, on our system a minimum method computation time of 300 milliseconds is necessary to obtain method granularities that produce reasonable parallel speedups.

Unfortunately, simple method computation times do not provide enough information for parallelization. For example, a method named `alpha()` can call another method `beta()`. The method computation time for `alpha()` includes the method computation time for `beta()`. Therefore, it is necessary to factor out the method computation time for `beta()` from the time for `alpha()` when considering whether to parallelize `alpha()` or not. This is especially important if the receiver objects for `alpha()` and `beta()` are different, since a decision must be made about which thread (or processor) to associate with each object. To analyze such situations, we define the *method active time* to be the time taken to execute the method code minus the active times for all of the methods that it calls. We define the *method aggregate time* as the total method computation time, the difference between method stop time and method start time. We also define the *object active time* to be the sum of the active times for all of the object's methods. Since many methods are called several times on an object, we also define *average method active* and *average method aggregate* times on each object. Finally, we express the average active method times and the active object times as a percentage of the total application time.

The major components of Revy are implemented using Java version 1.1, on Unix platforms, but also run on Windows and Mac OS computers. The PAS component for Java (called JavaSourcer) uses the JavaCC [7] library. The PAS component for C++ (called C++Sourcer) is implemented with the Solaris C++ compiler and uses the Sage++ [8] library. Revy does not have any unusual hardware requirements. More details about the design and implementation of Revy can be found in [9].

A Case Study - The 15-Puzzle

The benefits of Revy are best illustrated using a case study. The example is simple so that there are only a few objects and the details don't obscure the focus on the benefits of Revy. Examples with more complex object interactions can be found in [9]. In this paper, we apply Revy to a sequential program that implements the Iterative deepening A* (IDA*) search algorithm [10]. This algorithm is a search procedure for finding a maximum (minimum) cost solution in a directed graph. The search works by having a lower bound on the solution cost and then searching to prove or disprove the correctness of the bound. If the bound is incorrect, then it is incremented (decremented) and the search is restarted. This iterative algorithm is guaranteed to find the optimal solution given the assumption that all bounds are admissible. The IDA* algorithm can be used to solve many problems including game trees like Rubik's cube, as well as real applications like job-shop scheduling. In this paper we apply IDA* to solve the 15-puzzle. In the 15-puzzle we must move the tiles from the initial to the final configuration as shown in Figure 4.

Subtrees can be searched in parallel. A naive parallel version of the program would use a separate thread for each node object in the search tree. However, Revy quickly shows that such an approach would be disastrous. There is not enough granularity in the computation to justify a separate thread for each node object.



Figure 4. The 15-puzzle.

A more complex organization of parallel objects is necessary. Instead, the high level nodes in the tree should be programmed as sequential objects that share a thread. However, some of the nodes in the tree should be implemented as parallel objects, each with its own thread (frontier nodes). It will not actually create any node objects below it. Instead, it will use a recursive method to hold the state of the computation. However, before we actually parallelize the program we would like to know which nodes should be designated as frontier node objects. Some considerations include:

1. Spawning enough work to keep all processors busy.
2. Spawning work that is substantial enough to justify the overhead of the parallelism.
3. Ensuring that the size of the pieces of work results in good load balancing.

Each of these considerations can be addressed by Revy before the program is parallelized. The simplest approach is to invoke the parallelism at a fixed depth. For example, all depth 3 nodes can be designated frontier nodes. Revy gives information on the number of frontier nodes created. Figure 5 shows one of Revy's text views that lists all instances of the FrontierNode class, their unique object id numbers, their active object times and their relative percentages of active object time versus the active time of the application.

```
Sorted 24 objects by Active Time
-----
FrontierNode:5 - 29846 (11.67%)
FrontierNode:7 - 29009 (11.35%)
FrontierNode:10 - 27552 (10.78%)
FrontierNode:4 - 22137 ( 8.66%)
...
FrontierNode:3 - 4208 ( 1.65%)
FrontierNode:9 - 3679 ( 1.44%)
...
FrontierNode:22 - 705 ( 0.28%)
```

Figure 5. Active times (milliseconds) for each FrontierNode object.

A total of 24 objects are created when the master spawns work at depth 3 in the tree. This is enough to ensure that an 8-processor machine has enough active nodes to keep the machine busy. Although depth 2 has 10 frontier nodes, the number of nodes that are active enough for effective parallelization is smaller than 8, so some processors would quickly become idle.

However, this is not the whole story. Revy can also report the active times of individual methods. This is important since the active time can vary widely from call to call. For the 15-puzzle, it is not enough to generate enough pieces of work. Each piece of work must have sufficient granularity. From the Revy active method time data (Figure 6), one can see that the first four calls to the search() method do not have sufficient active time (less than 300 milliseconds), to make parallel calls reasonable even for the highest granularity instance of FrontierNode (object 5). For example, by the time we reach the 18th highest granularity object (object 9) the fifth call to the search() method does not even have sufficient granularity for parallelization.

Analysis of the search() method calls shows that there is an exponential distribution in the active times. In the early iterations of the algorithm, the active times are short. Each iteration increases the active time of a search() method call by as much as a factor of 10. Revy tells us that even though we create FrontierNode objects at depth 3 to get enough node objects to parallelize, the first four search calls to each of these FrontierNode objects should be done sequentially (on a single thread). Subsequent calls should be done in parallel (each on its own thread). Without measuring the granularity of

individual method invocations, such pre-parallelization analysis would be impossible.

```
6:search - 0 FrontierNode:5
20:search - 1 FrontierNode:5
42:search - 3 FrontierNode:5
66:search - 31 FrontierNode:5
90:search - 308 FrontierNode:5
114:search - 2948 FrontierNode:5
138:search - 26555 FrontierNode:5

3:search - 1 FrontierNode:9
16:search - 1 FrontierNode:9
38:search - 1 FrontierNode:9
62:search - 5 FrontierNode:9
86:search - 36 FrontierNode:9
110:search - 333 FrontierNode:9
134:search - 3302 FrontierNode:9
```

Figure 6. Active times (milliseconds) for the search() method for FrontierNode objects 5 and 9.

Figure 7 presents the same information in a graphical form. Revy has highlighted the objects whose active time percentages are greater than 10%. In addition, all method calls with active times greater than 2000 milliseconds are also annotated in the diagram. Whereas FrontierNode 7 has a method call with active time more than 26000 milliseconds, FrontierNode 17 has no method calls with active times as high as 2000 milliseconds. Of course these thresholds are adjustable by the user.

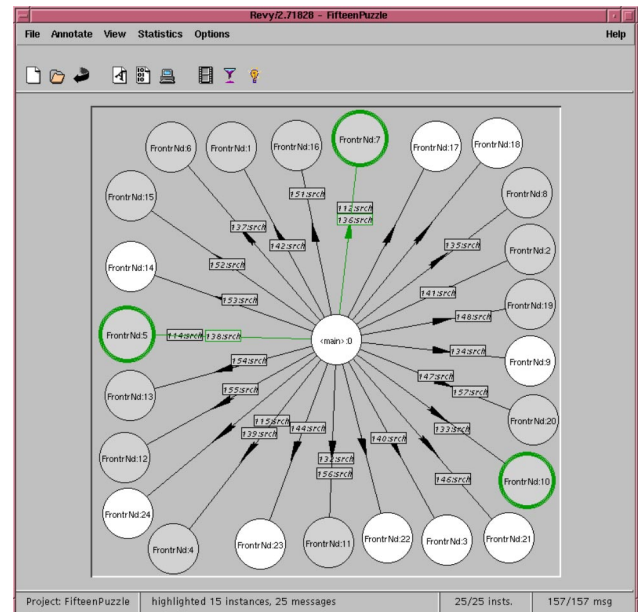


Figure 7. An object call graph for the 15-puzzle.

In order to test the Revy analysis, we wrote a parallel version of the IDA* implementation of the 15-puzzle. This application is parameterized by two parameters, d and p. FrontierNode objects are created at a fixed depth, d. However, the first parallel call to a frontier node occurs on the pth call, after p-1 sequential calls. Revy predicts that on an 8 processor machine, sufficient parallel

granularity occurs for depth, $d \geq 3$, on calls $p \geq 5$. This is a conservative estimate based on 300 millisecond minimum method times. In fact, it may be possible to get additional speed-ups for $d = 2$ or for $p < 5$ in some particular cases, if the particular puzzle selected happened to have good load balancing between frontier nodes.

We applied this suggested parallelization strategy to the 15-puzzle. We ran the parallel program on an SGI Challenge with eight 150-MHz R4400 processors and 384 Mbytes of memory. We obtained a speed-up of 4.4 on 8 processors. The time was 249 seconds for the sequential version and 56 seconds for the parallel version. The size of the speed-up is not the issue for this paper. The point is that Revy provides the information necessary to determine how to set parameters (like d and p in this application) to achieve the maximum speed-up possible for a given parallel implementation. Making more parallel calls (by selecting $p = 4$ or 3) decreases performance, due to low method granularity penalties (times of 67 and 61 seconds respectively). Making more parallel calls (by selecting $p = 6$ or $p = 7$) also decreases performance (times of 77 and 122 seconds respectively), due to lost opportunities for parallelism. Starting the parallelization at a smaller depth ($d = 2$ with $p = 5$) also decreases performance due to too few frontier nodes (124 seconds). Note that these correct predictions were made by analysing the sequential algorithm using Revy without writing any parallel code.

There is a wide variation in the active times for the different instances of the FrontierNode class. Some objects have larger searches than others (such as FrontierNode 5 compared to FrontierNode 9 in Figure 6). This distribution is a serious problem since it indicates that there will likely be poor load balancing, even using the advanced strategy of waiting for the fifth call to a frontier node to increase granularity. It is likely that $N-1$ processors may be idle waiting for the last (large) piece of work to complete. For our application, Revy suggests an even better solution, but it depends of the application keeping dynamic timing information about the search. Each leaf node in the master's tree can record the amount of time a `search()` method call takes and when it exceeds a threshold (some multiple of the minimum active time required needed for parallel tasks), then the master can expand that node to create more (smaller) pieces of work. In effect, the master now has a variable depth tree, where the deeper lines indicate more work. This type of analysis has been used for parallel alpha-beta search [11]. In this paper we have shown how Revy can quickly lead to the recognition that such a dynamic parallelization algorithm is necessary for maximum speed-ups on a particular problem.

Conclusions

In this paper we have introduced Revy, a tool that can use sequential analysis of object-oriented programs to determine parallelization strategies:

1. It provides visualization techniques that help users to understand the object communication patterns of their programs,
2. It provides profiling techniques for collecting and processing the method and object granularities to identify methods/objects for parallelization, and
3. It helps the user construct a parallelization strategy, even in the case of dynamic load-balancing problems.

Acknowledgements

This research was supported by grants from the Natural Science and Engineering Research Council of Canada.

References

- [1] D. Kafura and K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages", *ECOOP '89*, pp. 131-145, Cambridge University Press, 1989.
- [2] S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming" in G. Agha, P. Wegner and A. Yonezawa (editors), *Research Directions in Concurrent Object-Oriented Programming*, pp. 107-150, MIT Press, 1994.
- [3] P. America, "POOL-T: A Parallel Object-Oriented Language", in A. Yonezawa and M. Tokoro (editors), *Object-Oriented Concurrent Programming*, pp. 199-220, MIT Press, 1987.
- [4] Y. Yokote and M. Tokoro, "The Design and Implementation of Concurrent Smalltalk", *OOPSLA '86*, pp. 331-340, 1986.
- [5] D. Caromel, "A General Model for Concurrent and Distributed Object-Oriented Programming", *Workshop on Object-Based Concurrent Programming, OOPSLA '88*, 1988.
- [6] J. Waldo, G. Wyant, A. Wollrath and S. Kendall, "A Note on Distributed Computing", *Technical Report SMLI TR-94-29*, Sun Microsystems, November, 1994.
- [7] SunTest - The Java Testing Unit of Sun Microsystems, "Java Compiler Compiler WWW Page", www.suntest.com/JavaCC, 1997.
- [8] Extreme Research Group of Indiana University, "Sage++ WWW Page", www.extreme.indiana.edu/sage, 1995.
- [9] W. Hui, "Visualizing Object/Method Granularity for Program Parallelization", *M.Sc. Thesis*, Department of Computing Science, University of Alberta, 1998.
- [10] R. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, 27(1):97-109, 1985.
- [11] Mark Brockington and Jonathan Schaeffer. "APHID: Asynchronous Parallel Game-Tree Search", *Journal of Parallel and Distributed Computing*, vol. 60, pages 247-273, 2000.