

# An Expert System for the Development of Efficient Parallel Code

Gabriele Jost<sup>1\*</sup>, Robert Chun<sup>2</sup>, Haoqiang Jin<sup>1</sup>, Jesus Labarta<sup>3</sup>, and Judit Gimenez<sup>3</sup>

<sup>1</sup> NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA  
{gjost,hjin}@nas.nasa.gov

<sup>2</sup> Computer Science Department, San Jose State University, San Jose, CA 95192, USA  
Robert.Chun@sjsu.edu

<sup>3</sup> European Center for Parallelism in Barcelona-Technical University of Catalonia (CEPBA-UPC), cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain  
{jesus,judit}@cepba.upc.es

## Abstract

*We have built the prototype of an expert system to assist the user in the development of efficient parallel code. The system was integrated into the parallel programming environment that is currently being developed at NASA Ames. The expert system interfaces to tools for automatic parallelization and performance analysis. It uses static program structure information and performance data in order to automatically determine causes of poor performance and to make suggestions for improvements. In this paper we give an overview of our programming environment, describe the prototype implementation of our expert system, and demonstrate its usefulness with several case studies.*

## 1 Introduction

During the last decades large amounts of time and money have been spent on the development of large-scale scientific applications exposing an insatiable appetite for floating point operations per second. High performance computer architectures have evolved to satisfy this demand. Often the structure of the existing codes can not fully exploit the parallelism provided by the hardware. Considering the enormous investment in these codes and the niche market status of scientific applications, there is a strong incentive not to re-implement the applications from scratch, but rather, employ a conversion process to produce versions of existing codes optimized for the current most powerful computer architecture. Porting codes from one system to another usually requires substantial knowledge of both the underlying architecture and the applications themselves.

Current programming models support distributed memory, shared memory, and clusters of shared memory architectures. An example for the support of distributed memory programming is MPI [11] which provides the functionality for process communication and synchronization and assumes a private address space for each process. OpenMP [13] was introduced as an industrial standard for shared-memory programming with directives. Parallel programming on a shared memory machines can take advantage of the globally shared address space. Compilers for shared

---

\* The author is an employee of Computer Sciences Corporation.

memory architectures usually support multi-threaded execution of a program. OpenMP allows to exploit loop level parallelism by using compiler directives. Multiple levels of parallelism can be exploited by combining MPI and OpenMP.

The techniques described in this paper are applicable to many parallel programming paradigms, but we will restrict our discussion to OpenMP parallelization for shared memory computer architectures. During the program development the programmer will usually go through several cycles of placing directives into the code and gathering performance analysis data to check the efficiency of the generated code. Deciding which loops can and should be parallelized requires:

- static source code analysis to determine whether prohibitive data dependences exist, and
- dynamic performance information in order to determine time consuming code fragments that might benefit from parallelization.

For large scientific applications, the static and dynamic analysis results are typically very complex and their examination can pose a daunting task for the programmer.

At NASA Ames a programming environment for scientific applications is being developed which allows the user to jointly navigate through program structure and performance data. Components of the environment are the CAPO [6] parallelization tool and the Paraver [15] performance analysis system. Recently an expert system was added to the existing environment, which automates the preliminary examination of certain performance characteristics and their correlation to the program structure. In analogy to medical systems, the system automatically compares relevant performance metrics against threshold values, diagnoses the reasons for poor performance and suggests a therapy, if possible.

The rest of the paper is structured as follows: The components of the programming environment are introduced in Section 2. Section 3 describes the prototype implementation of the expert system. The usefulness of the expert system is demonstrated in section 4 by several case studies. Section 5 discusses some related work and the conclusions are drawn in section 6.

## **2 The Computer Aided Parallelization Process**

When parallelizing an application using OpenMP the program developer will usually carry out several cycles of placing directives into the code, instrumentation of the code for tracing, and performance analysis. In this section we will describe the components of our program environment which support the user in this task.

### **2.1 The CAPO Parallelization Support Tool**

CAPO was developed to automate the insertion of OpenMP directives with minimal user interaction. This is achieved by use of extensive interprocedural analysis from CAPTools [4] (now known as ParaWise) developed at the University of Greenwich, which provides a fully interprocedural and value-based dependence analysis engine. Dependence analysis results and other static program structure information are stored in an application database. Details on the CAPO parallelization process can be found in [6]. CAPO performs the following steps to exploit loop level parallelism:

- 1) Identify parallel loops and parallel regions based on dependence analysis.
- 2) Merge parallel regions and use the NOWAIT clause on successive parallel loops if possible, in order to reduce synchronization overhead.

- 3) Place OpenMP directives including variable scope declarations and perform necessary code transformations.

CAPO is currently integrated in the CAPTools environment and, thus, can access many features provided by CAPTools. This enables CAPO to provide a high amount of flexibility for the user to examine the program structure and provide information for the automated parallelization process in order to obtain highly efficient parallel code.

## 2.2 The Paraver Performance Analysis System

We have included the Paraver [15] performance analysis system in our programming environment. The Paraver system is being developed and maintained at the European Center for Parallelism in Barcelona (CEPBA) and supports performance analysis of a wide variety of programming paradigms such as MPI, OpenMP, and hybrid methods. It has two major components: A tracing package and a graphical user interface to examine the traces.

Paraver has a simple but very flexible trace format. Trace files from a variety of tracing packages can be converted to the Paraver format in order to leverage the power of the analysis tool. Paraver also provides its own tracing package, OMPITrace [12]. Depending on the computer system, OMPITrace dynamically instruments parallelization runtime libraries such as MPI and OpenMP. Examples of OpenMP related information dynamically instrumented and traced on our development platform (SGI Origin 3000) are:

- entry and exit of OpenMP runtime library routines,
- entry and exit of parallel loops,
- thread state such as running, idle, synchronization, or in fork/join overhead, and
- hardware counters.

User routines are not automatically traced on the SGI Origin, but OMPITrace provides library routines for manual source code instrumentation by the user.

The trace collected during the execution of a program contains a wealth of information, which as a whole can be overwhelming. The information must be filtered to gain visibility of a critical subset of the data. This can be done through timeline graphical displays or by histograms and other statistics. Paraver provides flexibility in composing displays of trace data. Examples are timeline views to show the particular state that a thread is in, when parallel functions are being executed by each thread, or what the MIPS or cache miss rates are for a given time interval. The Paraver analysis module allows the computation of a given performance metric from the records in the trace. Paraver provides flexibility in composing displays or the calculation of metrics that are suitable for a particular problem. A user can specify through the Paraver GUI a certain view of a trace files or define how to compute a given performance index from the trace. The specification can be saved to a configuration file and re-used later on. This feature allows the programmability of performance metric computations, which is essential for the automation of performance analysis.

## 2.3 User Interaction with the Programming Environment Components

During the program parallelization cycle the user will typically go through several CAPO and Paraver sessions. The first step is usually the generation of an application database. An important item of information contained in the application database is a list of questions about values of

program variables. These are questions that arose during the dependence analysis, often due to the fact that the values of certain variables are not known at the time of the analysis. Unresolved questions indicate conservatively assumed dependences which can prevent parallelization. The user may browse the list of questions and provide assertions to make a more precise dependence analysis possible. Removing conservatively assumed dependences can greatly

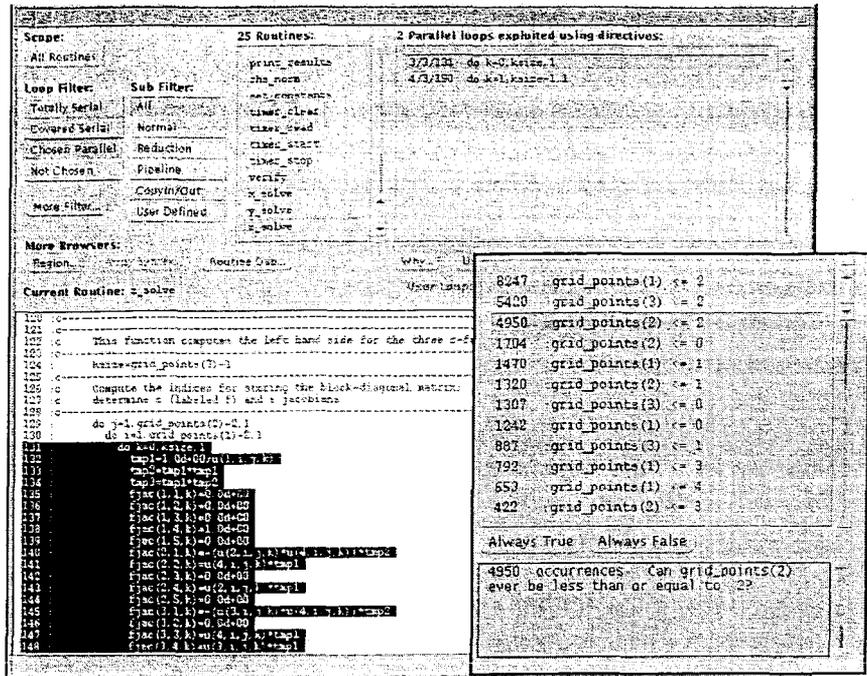


Figure 1: CAPO Directive Browser and Knowledge Database

increase the efficiency of the generated code.

After providing user knowledge about runtime values of variables the user can repeat the analysis and let CAPO generate OpenMP directives. CAPO provides a directive browser which is designed to display information gathered during the parallelization such as the reasons for loops to be parallel or serial, and the relevant variables. A snapshot of the CAPO directive browser and a list of questions from the knowledge database is shown in Figure 1. For each subroutine the user can retrieve information about the loops it contains. CAPO classifies loops by types such as serial, parallel, or being part of a loop nest containing parallelized loops. The browser also provides user interfaces to declare certain variables as shared or private and to overwrite the automatically determined type. In order to determine whether certain variables can be made shared or private, the user has the possibility to examine individual edges of the dependence graph and inquire about the reasons for their existence. The user can explicitly remove edges from the dependence graph. Using the CAPO directives browser to declare the scope of a variable will automatically remove corresponding dependence

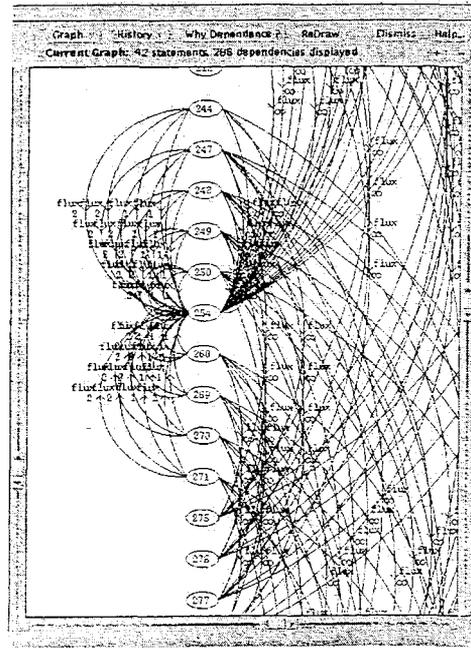


Figure 2: Snapshot of session browsing the dependence graph.

edges which simplifies the process of dependence pruning for the user. A snapshot of the dependence graph browser is shown in Figure 2.

After pruning the dependences and generating OpenMP directives the user will typically run the application with a representative set of input data and analyze the performance of the generated code. Paraver allows the user to obtain a qualitative view about the program and quantify the behavior by calculating numerical statistics. For example it is often necessary to determine how much time the individual threads spent in different parallel regions. A sample view and statistic are displayed in Figures 3 and 4.

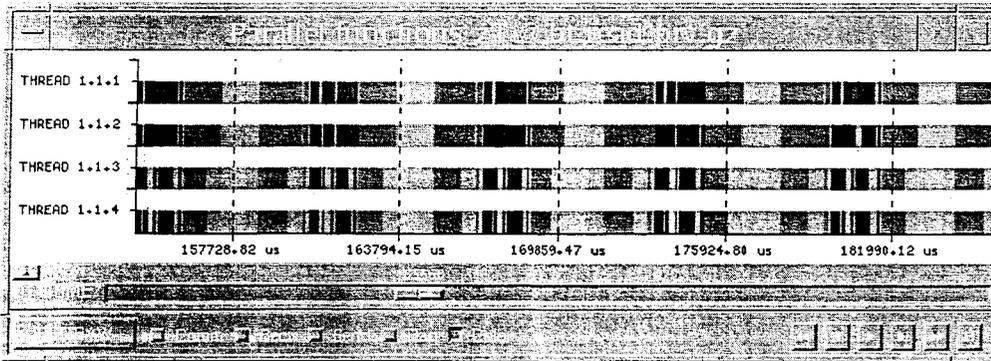


Figure 3: Paraver view of time spent in the parallel regions in an OpenMP code running on 4 threads. The different shadings indicate time spent in different parallel regions

Both tools, CAPO and Paraver, provide the user with capabilities and information that can assist him in accomplishing code porting from a serial machine to a parallel machine; however, the user must take an active role in driving the analysis tools and in interpreting their results to make the proper code transformations to improve parallelism. Both tools provide a high degree of flexibility and generate a wealth of information. While this is essential for efficient parallelization of an application, there is also a drawback: For large scientific codes, the static and dynamic analysis data can be quite complex, and a high level of knowledge and expertise is required by the user to comprehend its meaning. The process of filtering essential from non-essential information can be quite time consuming. For example, pruning the information shown in Figure 2 can be very challenging. Likewise, a thorough dependence analysis will require many examinations of the kind depicted in Figures 3 and 4. In addition to that, because CAPO and Paraver are separate tools, correlating information between them is something the user must also perform manually. The primary goal of our project is to assist the user in the process-intensive and knowledge-intensive code parallelization task to make it easier and more efficient. From a cost-performance point of view, the objective is to assist the user in obtaining maximum code performance gain with a minimum amount of analysis and transformation time. The idea is to unburden the user as much as possible by helping him to focus on only the relevant information from the static and dynamic analysis. This should in turn help the user to focus on the code with the best potential for speedup via parallelization.

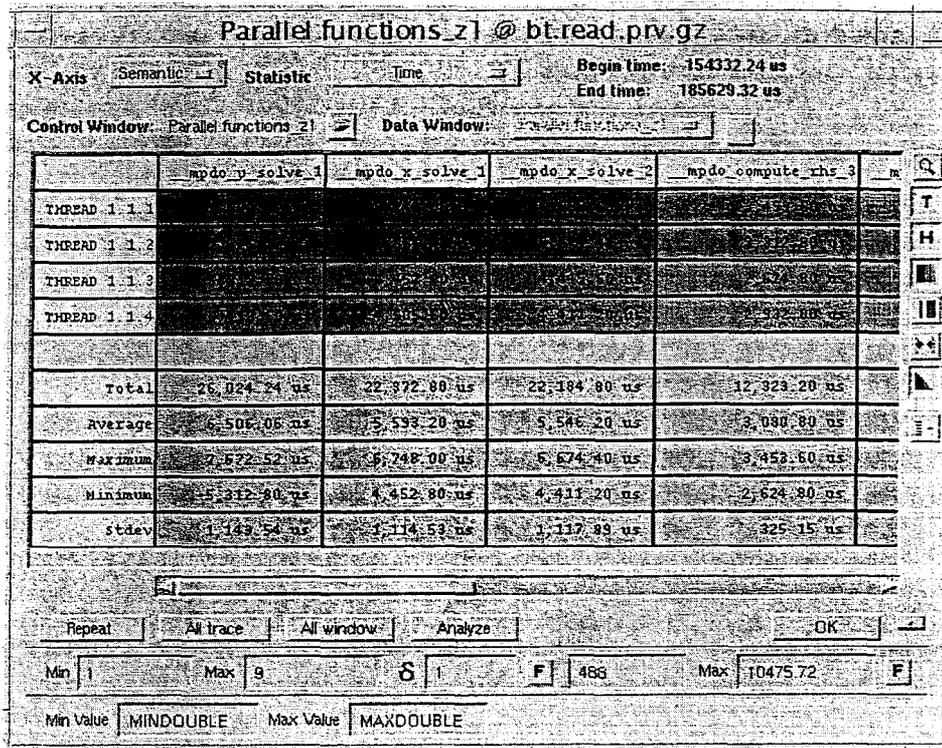


Figure 4: Paraver analysis of the timeline view displayed in Figure 3. The chosen statistic is the time spent by the individual threads in different parallel regions. Darker shading indicates higher values.

### 3 The Expert System Prototype Implementation

A computerized parallelization advisor was developed using an expert system implemented in CLIPS ("C" Language Integrated Production System) [3]. CLIPS is an open source expert system development toolkit developed at the Johnson Space Center and has been used in numerous NASA applications. The expert system was inserted into the existing tool infrastructure to preprocess the raw static and dynamic analysis information. There are several advantages to this architecture involving the expert system. First, the expert system will work with, rather than replace either the user or the existing toolset. This facilitates a non-invasive insertion of the new technology. The user can make use of the expert system advice, but can also choose to work as he has done previously. Second, the expert system approach enables the system to be programmed using a set of rules input into the knowledge base of CLIPS. Because the rules are modular and stored separately from the expert system inferencing mechanism, they can be modified or added incrementally to the system. Third, the effective parallelization of serial code requires broad knowledge about the tools, the target computer architecture, code parallelization techniques, and the algorithms used in the source code. Capturing this knowledge in the expert system will enable the information to be reused.

By performing information fusion on the static and dynamic analysis, the expert system can help the user to filter, correlate, and interpret the data. Relevant performance indices are automatically calculated and correlated with program structure information. We have also added an

automatic selective source code instrumentation module to facilitate the use of the expert system. The overall architecture of our program environment is depicted in Figure 5. In the following subsections we describe its various components.

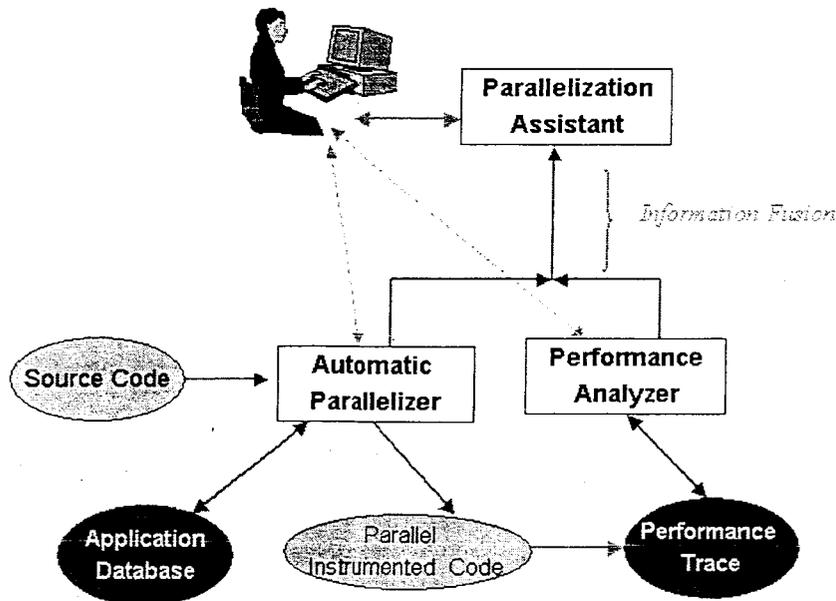


Figure 5: Architecture of the expert system programming environment. The Parallelization Assistant Expert System fuses program structure knowledge and performance trace information to aid the user narrowing down performance problems. The user can still interact directly with the parallelization and performance analysis tools for fine tuning of the applications performance.

### 3.1 Selective Source Code Instrumentation

Conducting performance analysis requires one or more trace files that contain runtime information about the program behavior. In our program environment we use the OMPItrace package to collect such traces. The OMPItrace module dynamically instruments certain runtime libraries, but does not automatically trace entry and exit to user routines. It may not always be desirable to trace all of the subroutines because of large instrumentation overhead. For critical code segments, on the other hand, it is often desirable to obtain information on a finer granularity than a subroutine call. We have extended the source code transformation capability of CAPO to automatically insert calls to the OMPItrace library. In addition, we use the program structure information from the application database in order to determine which parts of the source code should be traced. In our prototype we use the following simple heuristics to select instrumentation points:

- routines that are not contained within a parallel region or a parallel loop and contain at least one DO-loop, and

- outermost serial loops.

Parallelized loops and parallel regions are automatically traced by the OMPItrace package. More details on the automatic selective instrumentation can be found in [7].

### 3.2 Automatic Retrieval of Performance Indices

We have extended the Paraver system by Paramedir, a non-graphical user interface to the analysis module. Paramedir accepts the same trace and configuration files as Paraver. This allows capturing the same information in both systems. The detailed human driven analysis can thus be translated into rules suitable for processing by an expert system. Complex performance metrics, determined by a human expert, can be automatically computed and processed. In a first step, the expert system invokes Paramedir to build up a table of performance metrics. In the prototype implementation of our system we consider a small number of metrics relevant to OpenMP parallelization. We determine where the time is being spent on a loop or subroutine level and then evaluate whether the time is spent efficiently. Instrumented routines are not called from within a parallel region, due to our instrumentation strategy. The Paraver analysis module is used to calculate the following metrics.

- **Time profile:** The first step is the calculation of a time profile on the basis of the automatically instrumented routines and loops. We calculate the percentage of time spent in the instrumented code segments. We are considering the exclusive time of the instrumented routines. By this we mean that when routine `sub_a` calls routine `sub_b`, and both of them are instrumented, the time reported for `sub_a` will not include the time for `sub_b`.
- **Efficiency:** To evaluate the parallel efficiency of a particular code segment we compute the average utilization of all threads. As explained in Section 2.2 the trace file contains information about the state of a thread. If we denote by *TotTime* the sum of the times in running state over all threads, excluding all OpenMP overhead, by *ElapsedTime* the elapsed time of the routine and by *nt* the number of threads, then the ratio  $(TotTime)/(Elapsed\ Time * nt)$  gives a measure for the parallel efficiency. A value close to 1 indicates good efficiency. A value close to  $1/nt$ , on the other hand, indicates that only the master thread performed useful computations most of the time.
- **Granularity:** We determine the average duration of a parallel construct. As explained in Section 2.2, entry and exit of compiler generated routines containing the body of parallel loops are traced dynamically and do not require source code instrumentation.
- **Sequential section:** If we denote by *SeqTime* the time the master thread spends outside of parallel constructs and *TotTime* the sum of the running time over all threads, then the ratio  $SeqTime/TotTime$  is a measure for the sequential fraction within each routine. A value close to 1 indicates that most of the time is spent running sequentially.
- **Load balance:** We use the standard deviation in the fraction of useful time for each thread as an indicator for the load balance within each routine or loop. At this point we can not determine whether the load imbalance is caused by an unbalance in the amount of work in terms of instructions or by memory access time. In the future we will take hardware counters and different scheduling strategies into consideration.

### 3.3 Information Fusion and Rule Based Optimization Guidance

After building the table containing the performance indices, the expert system augments this table by adding program structure information from the application database. At this point our main interest is the optimal placement of OpenMP directives. We are therefore mainly interested in the information that would be displayed in the CAPO directives browser described in Section 2.1. In our prototype implementation we only retrieve the type of the instrumented code segment, such as loop or subroutine, the CAPO loop identifier, and the type of the loop.

The expert system now applies a set of rules to the values contained in the augmented table in order to point the user to performance problems. If possible the user will also be presented with suggestions for improvements. With the first set of rules we identify routines and loops in the code that take a large percentage of the execution time and show low efficiency. Using Amdahl's Law, the expert system gives a limit for the number of threads that can be used efficiently. A second set of rules determines the cause for the inefficient parallelization, such as large sequential sections, fine grained parallelization, or load imbalance. In order to determine the cause of inefficient parallelization the expert system compares the performance metrics against established threshold values. The threshold values are based on prior experiences. A third set of rules checks the loop type and, if appropriate, suggests possibilities for more efficient parallelization. A sample rule is shown in Figure 6.

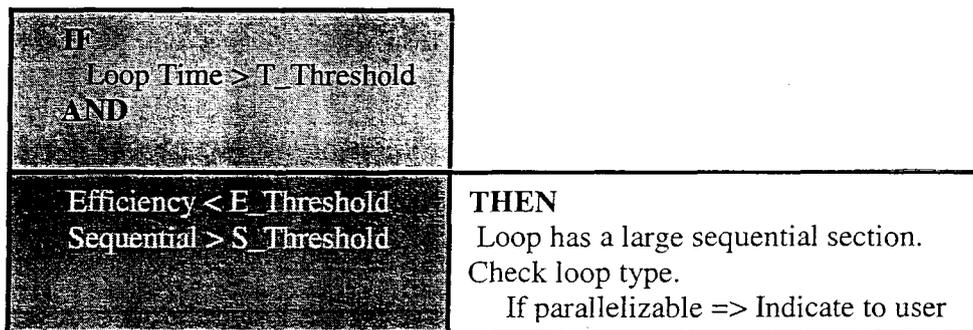


Figure 6: Example rule from the expert system which is applied to sequential outer loops

## 4 Case Studies

We have tested the parallelization assistant expert system on benchmark codes from the field of computational fluid dynamics (CFD) and a module from a full scale climate modeling application.

### 4.1 NAS Parallel Benchmarks BT and FT

The NAS Parallel Benchmarks [2] were designed to compare the performance of parallel computers on CFD applications. For our evaluation we used the implementation described in [5], not including the OpenMP directives.

The BT benchmark solves three systems of equations resulting from an approximate factorization that decouples the  $x$ ,  $y$  and  $z$  dimensions of the 3-dimensional Navier-Stokes equations. These systems are block tridiagonal consisting of  $5 \times 5$  blocks. Each dimension is swept sequen-

tially. The time consuming routines are the solvers in each of the spatial dimensions and the calculation of the right-hand side. In this scenario we assume that the user has already inserted some OpenMP directives himself and now wants to optimize the code using CAPO. The user supplies all information about input values to answer the questions from the knowledge database. This results in a very precise dependence analysis. The code is then automatically instrumented based on the directives inserted by the user. A time line view for 4 threads is shown in Figure 7.

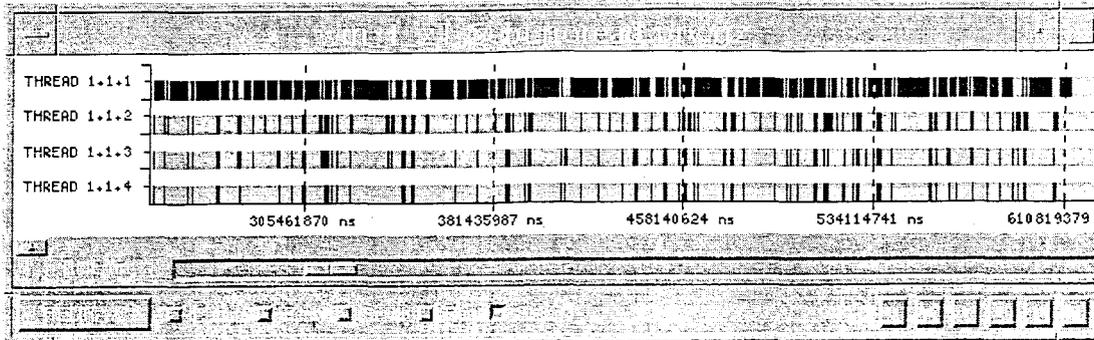


Figure 7: Time line view of the useful thread time for the BT benchmark running on 4 threads. Dark shading indicates time spent in calculations, light shading indicates idle or synchronization time. The view shows large sections of non-useful time

The time line shows large sections of non-useful time for the slave threads. The master thread shows a higher amount of useful time, but the expert system can determine that this is not spent within parallel regions. The analysis for one of the time consuming loops is shown in Figure 8.

```

Routine z_solve Loop Number 1
****
**** Takes 21.41 % of the execution time.
**** Runs with 37.0 % efficiency on 4 threads.
**** The theoretical limit on the number of threads
      for an efficiency of 75.0% is 1 thread.
****   Loop has a large sequential section
****   Loop contains fine grained parallel loops.
===>>> POSSIBILITY FOR OPTIMIZATION:
===>>> The loop has not been parallelized by the user.
===>>> The loop is parallelizable according to CAPO.

```

Figure 8: Expert system analysis of a time consuming loop with BT. The rule applied for this case is shown in Figure 6

The analysis results quantify what we could qualitatively derive from the time line view. The parallel efficiency for 4 threads is low and it is indicated that the code would run faster when employing only 1 thread. A large amount of time is spent in routine z\_solve in loop number 1. The loop number corresponds to the one generated by CAPO. The expert system displays CAPO analysis results indicating that the loop could be run in parallel. The indicated loop number is

generated by CAPO. It allows the user to identify the loop in the CAPO directives browser and incrementally insert the directives. After parallelization of all of the loops indicated by the expert system and re-running the analysis, this time on 64 threads, the user obtains the report shown in Figure 9.

```

Routine z_solve
****
**** Takes 26.45 % of the execution time.
**** Runs with 92.0 % efficiency on 64 threads.

```

Figure 9: Expert system analysis run for the optimized code running on 64 threads

The FT benchmark is the computational kernel of a spectral method based on a 3D Fast Fourier Transform (FFT). We assume the scenario that the user parallelizes the code using CAPO, without pruning the dependence graph. The result is that innermost loops are chosen for parallelization. A Paraver timeline view displaying the running time of the threads is shown in Figure 10. The timeline view indicates a large amount of short bursts of running time, resulting from the fine grained parallelization. The expert system report is shown in Figure 11.

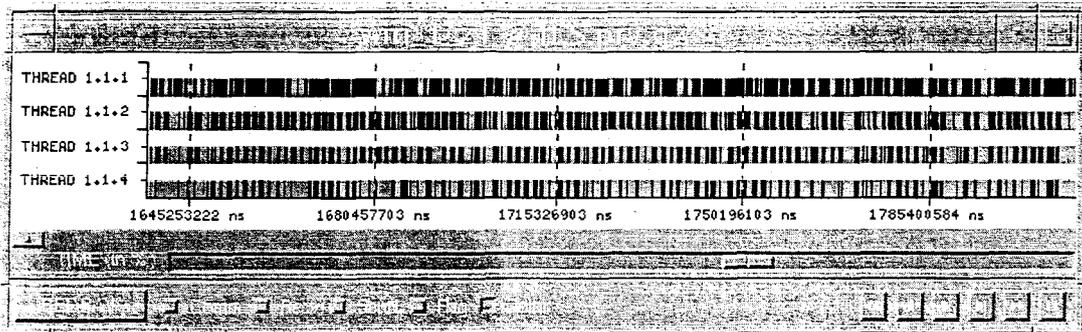


Figure 10: Timeline view of the FT benchmark on running on 4 threads. Dark shading indicates time spent in calculations, light shading indicates non-useful time. The view shows fine-grained sections of useful calculations

```

==> Routine fftz2
****
**** Takes 78.66 % of the execution time.
**** Runs with 55.0 % efficiency on 4threads.
**** Routine contains fine grained parallel loops.
**** Most time consuming parallel regions:
*
*****>>> __mpregion_fftz2_1 avg. duration 124 us
*
==>> NECESSARY ACTION:
==>> Try moving parallelization to an enclosing loop.
==>> Use CAPO's directive browser to examine dependences.

```

Figure 11: Expert system analysis for the FT benchmark

The system determines that in this case the problem is not due to large sequential sections, but rather that the parallelization granularity is too fine, with an average duration of less than 150 micro seconds per execution of the parallel construct. The system advises to investigate the possibility to move the parallelism to an enclosing loop. At this point we do not indicate a list of candidates for loops that should be examined. We are currently working on integrating nesting information about instrumented code segments in order to point the user to enclosing loops.

#### 4.2 The PSAS Conjugate Gradient Solver

The PSAS Conjugate Gradient Solver is a component of the Goddard EOS (Earth Observing Systems) Data Assimilation System. It contains a nested conjugate gradient solver implemented in Fortran 90. The time consuming portions of the code are a symmetric covariance multiply and the first level conjugate gradient solver. As in the case of the BT benchmark, we assume the scenario where the code has been parallelized by the user. The user then generates an application database and the code is automatically instrumented based on the existing directives. A time line view of the useful computations for a run on 4 threads is shown in Figure 12. We can visually detect a load imbalance among the threads. The reason for the load imbalance is due to the fact that the iterations of the nested conjugate gradient solver greatly vary in time. A load imbalance also exists in the case of BT (see Figure 7). For BT the load imbalance is introduced by the large sequential sections, which yield the effect that only the master thread performs useful calculations for a large amount of time. For the case of PSAS, the expert system detects that there are no large sequential sections and that the granularity of the parallelization is not below the established threshold. A potential cause for the load imbalance is the variance in the average duration of the loop iterations. The reported analysis is displayed in Figure 13.

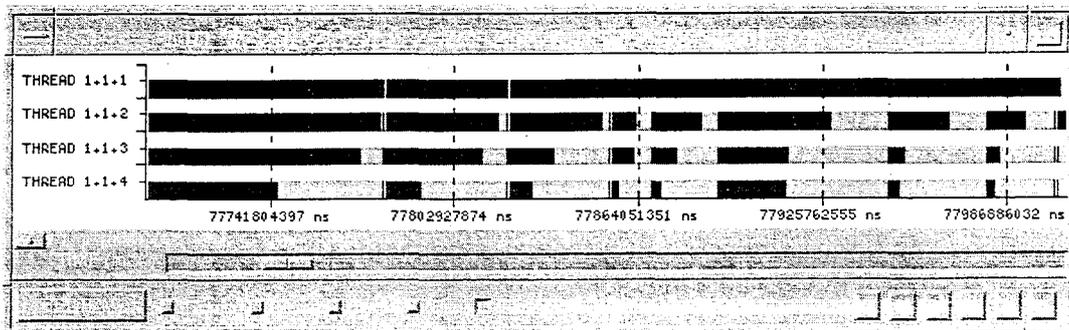


Figure 12: Timeline view of the PSAS Conjugate Gradient solver running on 4 threads. Dark shading indicates that the threads are performing useful calculation. The view shows a load imbalance in useful time

The user can now examine CAPO loop number 5 subroutine in `sym_Cxpy` in the directives browser. The loop number is generated by CAPO and allows the user to identify the loop in the CAPO directive browser. Using dynamic instead of static scheduling increased the performance of the test case under consideration by a factor of 2 when 8 threads are employed. At this point we need to point out that dynamic scheduling should be applied with caution. In the current case

```

==> Routine sym_Cxpy Loop Number 5
****
**** Takes 75.75 % of the execution time.
**** Runs with 46.0 % efficiency on 4 threads.
**** The loop contains parallelized loops
      which show load imbalance!
***
==>> POSSIBILITY FOR OPTIMIZATION:
==>> Check the parallelized loops within the indicated loop
==>> Consider adding the SCHEDULE(DYNAMIC)
      clause for better work load distribution

```

Figure 13: Expert system analysis of a time consuming routine with the PSAS Conjugate Gradient Solver

it is beneficial since the workload among the iterations varies widely and each of the iterations provides a large enough workload. In case of dynamic scheduling our system can derive a histogram of the individual iterations and determine their variation. In case of static scheduling, however, we can not draw any conclusion about the duration of an individual iteration. We are considering emitting information about the iteration space into the trace. Furthermore, we will, in the future, include the analysis of traces containing hardware counter information. This will enable us to investigate whether the imbalance in computation time is due to an imbalance in the number of instructions or memory access time.

## 5 Related Work

There are number of commercial and research automatic parallelization and performance analysis tools that have been developed over the years. We can only name a few.

The SUIF Explorer [8] developed at Stanford University is an interactive parallelization tool based on the SUIF [16] compiler. It performs extensive static dependence analysis and also includes a set of dynamic analyzers to provide runtime information. Runtime information includes checking dependences and time profiles for loops and routines. The user can provide assertions via a graphical user interface. The most notable difference to our approach is that by interfacing to a full performance analysis system like Paraver, we have more performance metrics available than just execution time. This allows us a more flexible and detailed analysis and greater opportunity to detect performance problems.

The URSA MINOR [14] tool, developed at Purdue University, also aims at using performance data and program structure information in order to guide the user through the optimization process. It includes a performance advisor which automatically checks common performance bottle necks. Our approach differs in that we are interfacing to existing parallelization and performance analysis tools and have the full functionality of these tools available. Also, to our knowledge the URSA MINOR tool does not include a rule based expert system.

An example of a commercial product for performance analysis is Vampir [18] which allows tracing and analysis of MPI codes. The CAPTools system provides an interface for generating

VAMPIR traces and invoking VAMPIR from within CAPTools for performance analysis of message passing applications. We are using the Paraver system in our programming environment because it provides great flexibility in the calculation of performance metrics and allows their automatic retrieval via a non-graphical user interface.

Two research projects on performance analysis are Paradyn [9], which is developed at the University of Madison, and Aksum, which is part of the ASKALON [1] project conducted at the University of Vienna. Both aim at the automatic detection of performance bottlenecks. A general performance model for OpenMP is proposed in [10]. Performance libraries based on this model have been developed for the TAU (Tuning and Analysis Utility) performance analysis framework ([10], [17]) and the EXPERT automatic event trace analyzer [19].

Our approach differs from the projects above in various ways. In addition to performance trace data we also use dependence analysis information in order to detect performance problems. We do not focus on a set of bottlenecks, but rather examine the time consuming sections of the code according to rules which a user would apply. The rules are modular and can be modified or added incrementally to the system.

## 6 Conclusions and Future Work

We have built the prototype of an expert system to assist the user in the development of efficient OpenMP code. The system was integrated into our parallel programming environment. It uses the static program analysis information available from the CAPO parallelization tool and the performance trace data and statistical analysis module from the Paraver performance analysis system. This information is correlated, fused and passed to the expert system for analysis. We have demonstrated in several case studies how the expert system analysis results can guide the user to efficient OpenMP parallelization.

A thorough performance analysis will usually require more than one trace file. The immediate plan for our prototype implementation is to allow the analysis and comparison of several traces for the same application. This will include the comparison of trace files with and without hardware counters, traces employing different numbers of threads, and different scheduling algorithms.

At the moment the information used from the application database is very minimal. We plan to include further information such as obstacles to parallelization and the reasons for their existence. At this point the expert system points the user to the critical parts of the code, but then the CAPO directives browser has to be used for further examination of dependences.

Not all performance problems in full-scale applications will be solved automatically. At this point it is still not clear where to draw the line between automated and user guided responsibilities. Experiments with our prototype will allow identification of more tasks that can be automated.

## Acknowledgments

This work was supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/AMTI, by the NASA Faculty Fellowship Program, and by the Spanish Ministry of Science and Technology, by the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA). We

thank the CAPTools development team for many helpful discussions and the continued support of our work.

## References

- [1] ASKALON, <http://www.par.univie.ac.at/project/askalon/>
- [2] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.
- [3] CLIPS: A Tool for Building Expert Systems, <http://www.ghg.net/clips/CLIPS.html>.
- [4] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. <http://www.parallelsp.com/parawise.htm>.
- [5] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance," *NAS Technical Report NAS-99-011*, 1999.
- [6] H. Jin, M. Frumkin and J. Yan. "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," *Proceedings of Third International Symposium on High Performance Computing (ISHPC2000)*, Tokyo, Japan, October 16-18, 2000.
- [7] G. Jost, H. Jin, J. Labarta, J. Gimenez, "Interfacing Computer Aided Parallelization and Performance Analysis," *Proceedings of the International Conference of Computational Science - ICCS03*, Melbourne, Australia, June 2003.
- [8] Liao, S., Diwan, A., Bosch, R. P., Ghuloum, A., Lam, M., "SUIF Explorer: An interactive and Interprocedural Parallelizer," *7<sup>th</sup> ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Atlanta, Georgia, (1999), 37-48.
- [9] B.P. Miller, M.D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithapdam and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer* 28, 11, pp.37-47 (1995).
- [10] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," *Internal Report FZJ-ZAM-IB-2001-09*, Research Centre Juelich, Germany, 2001.
- [11] MPI 1.1 Standard, <http://www-unix.mcs.anl.gov/mpi/>.
- [12] OMPItrace User's Guide, [http://www.cepba.upc.es/paraver/manual\\_i.htm](http://www.cepba.upc.es/paraver/manual_i.htm).
- [13] OpenMP Fortran/C Application Program Interface, <http://www.openmp.org/>.
- [14] I. Park, M. J. Voss, B. Armstrong, R. Eigenmann, "Supporting Users' Reasoning in Performance Evaluation and Tuning of Parallel Applications," *Proceedings of PDCS'2000*, Las Vegas, NV, 2000.
- [15] Paraver, <http://www.cepba.upc.es/paraver/>.
- [16] SUIF Compiler System, <http://suif.stanford.edu/>.

- [17] TAU: Tuning and Analysis Utilities, <http://www.cs.uoregon.edu/research/paracomp/tau/>.
- [18] VAMPIR User's Guide, Pallas GmbH, <http://www.pallas.de/>.
- [19] Wolf, F., Mohr, B., "Automatic Performance Analysis of SMP Cluster Applications," Tech. Rep. IB 2001-05, Research Centre Juelich, Germany, 2001.