

An Expert Assistant for Computer Aided Parallelization

**G. Jost^{1*}, R. Chun², H. Jin¹,
J. Labarta³, J. Gimenez³**

¹NASA Ames Research Center

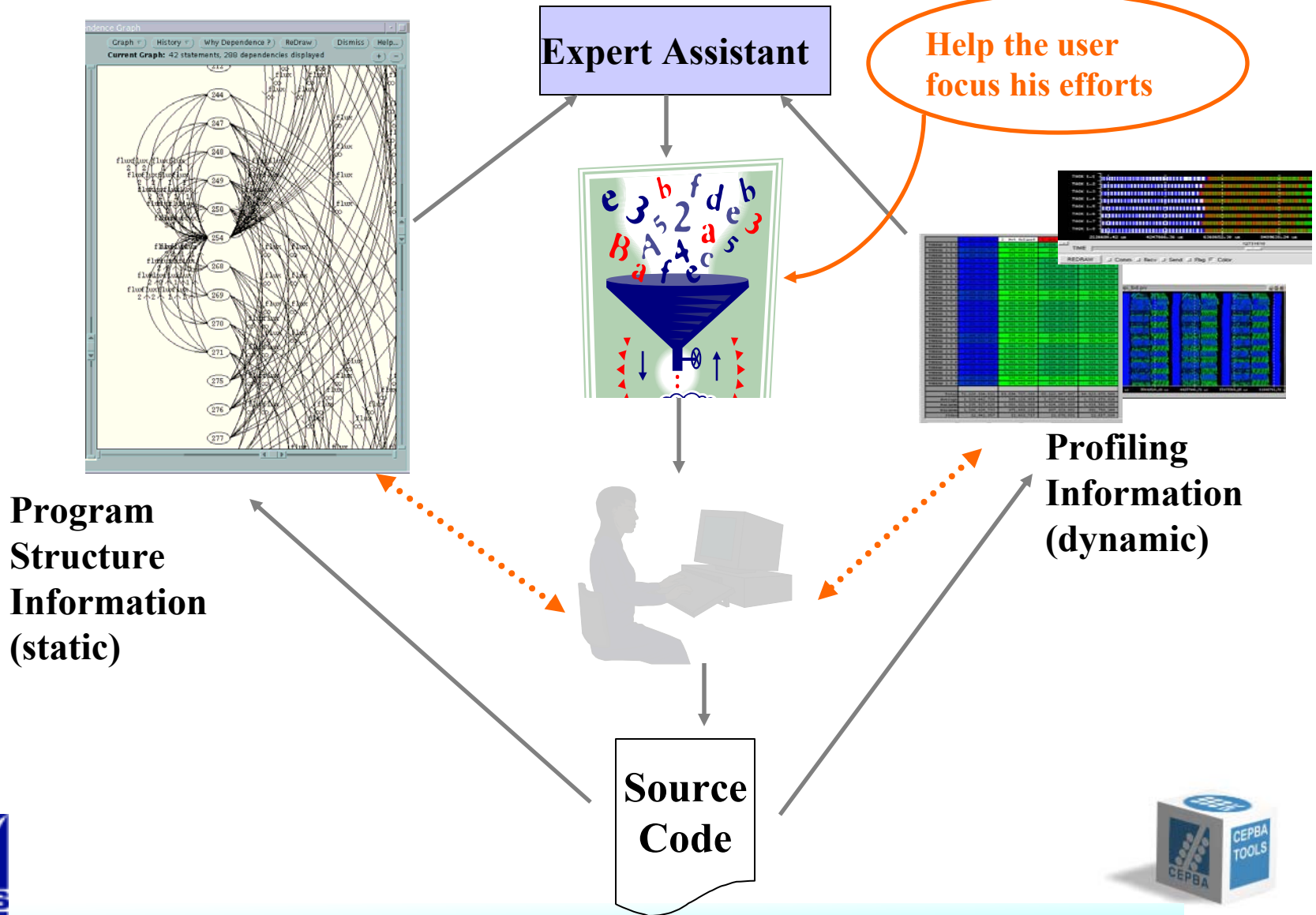
****Computer Sciences Corporation***

***²Computer Science Department, San
Jose State University***

***³European Center of Parallelism of
Barcelona (CEPBA)***

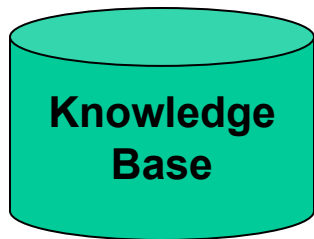


The Parallel Program Development Cycle



Rule-Based Paradigm

- **Rules are stored in a Knowledge Base**
- **Forward chaining (data driven):**
 - ▶ Infer a conclusion given all asserted facts
- **Backward chaining (goal directed):**
 - ▶ Gather facts given unsubstantiated hypothesis
- **Rules are selected for execution by an inference engine**



IF (`time_consuming`) AND (`not_efficient`)
THEN (`optimize`)



Prototype Implementation Components

- **The CAPO Parallelization Tool**
 - ▶ Static Program Structure Information
- **The Paraver Performance Analysis System**
 - ▶ Dynamic Performance Trace Data
- **The Paramedir Command-line Interface to Paraver**
 - ▶ Automatic Calculation of Performance Metrics
- **The ParaMed Expert System**
 - ▶ Correlation and Interpretation of Facts



The CAPO Parallelization Tool

- **Developed at the NASA Ames Research Center**
- **Dependence analysis module from ParaWise (aka CAPTools, University of Greenwich)**
- **Analysis results stored in application database**
- **Inserts OpenMP parallelization directives into sequential F77 or F90 code:**
 - ▶ Identify and define **PARALLEL** regions and loops
 - ▶ Identify variable scope (e.g. **SHARED**, **PRIVATE**)
 - ▶ Optimizations, e.g.:
 - Merge consecutive parallel regions
 - Detect and produce NOWAIT directives
 - ▶ Provides extensive set of browsers
 - ▶ Allows user interaction to improve efficiency of generated code



Parallelization Challenges

The Main Window

- Are the directives placed efficiently?
 - ▶ Many loops
 - ▶ Many dependences
 - ▶ Large knowledge data base

The image shows two overlapping windows from the CAPO tool. The top window, titled "CAPO: Directives Browser", displays a list of 36 routines and 8 covered serial loops. The bottom window, titled "CAPO: Why Directives?", provides a detailed view of a specific loop (7/4/158: DO 331 K=KS, KT, 1), showing its type as "Covered Serial" and the reason as "True dependence, containing parallel loops". It also lists anti-dependency variables (FFA, FFB, FFC, FFD, FFE, FFF, FFG, ffi), output-dependency variables (AE, AW, FFA, FFB, FFC, FFD, FFE, FFF), and in/output-dependency variables (>AE, >AW, <FFA, <FFB, <FFC, <FFD, <FFE, <FFF). A "Hints" section indicates the presence of 15 parallel loops and lists variables with loop-carried true and anti-dependence. A "Parallel loops" section lists several loop headers.

The Why Window

Dependence Graph



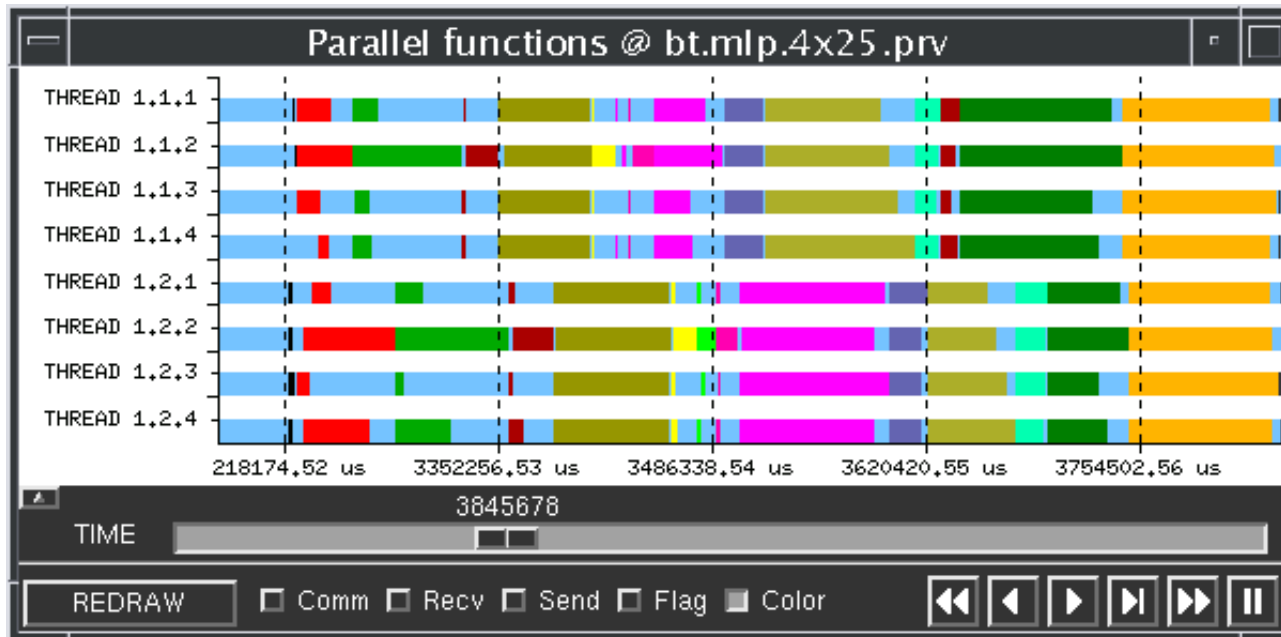
Paraver Performance Analysis System

- **Developed at CEPBA**
- **OMPItrace Package**
 - ▶ Tracing on thread, process, application level
 - ▶ Dynamic tracing of
 - Entry/exit parallel constructs
 - Entry exit parallelization runtime library calls (MPI, OpenMP)
 - Hardware counter
 - ▶ Also allows user instrumentation of source code.
- **GUI to visually examine traces**
- **Paraver Analysis Module**



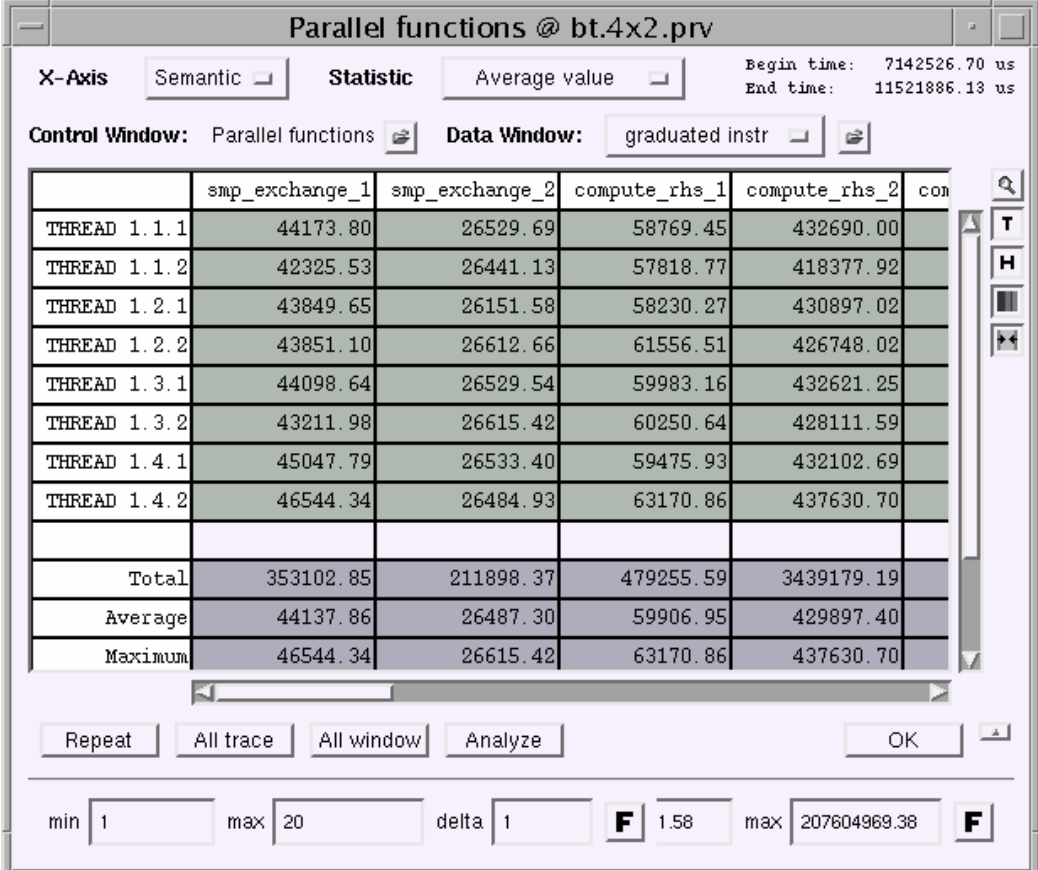
The Paraver GUI

- **Specific view of trace data:**
 - ▶ Composed using the Paraver GUI
 - ▶ Configuration saved to file for re-use
- **Example: Timeline view of parallel regions**
 - ▶ Parallel regions indicated by different colors



The Paraver Analysis Module

- Calculate statistics
- Display profile information as tables or histograms
- Correlate statistics with Paraver objects
- Calculation and display:
 - ▶ Composed by user via GUI.
 - ▶ Configuration saved to file for re-use



Parallel functions @ bt.4x2.prv

X-Axis: Semantic Statistic: Average value Begin time: 7142526.70 us End time: 11521886.13 us

Control Window: Parallel functions Data Window: graduated instr

	smp_exchange_1	smp_exchange_2	compute_rhs_1	compute_rhs_2	con
THREAD 1.1.1	44173.80	26529.69	58769.45	432690.00	
THREAD 1.1.2	42325.53	26441.13	57818.77	418377.92	
THREAD 1.2.1	43849.65	26151.58	58230.27	430897.02	
THREAD 1.2.2	43851.10	26612.66	61556.51	426748.02	
THREAD 1.3.1	44098.64	26529.54	59983.16	432621.25	
THREAD 1.3.2	43211.98	26615.42	60250.64	428111.59	
THREAD 1.4.1	45047.79	26533.40	59475.93	432102.69	
THREAD 1.4.2	46544.34	26484.93	63170.86	437630.70	
Total	353102.85	211898.37	479255.59	3439179.19	
Average	44137.86	26487.30	59906.95	429897.40	
Maximum	46544.34	26615.42	63170.86	437630.70	

Repeat All trace All window Analyze OK

min 1 max 20 delta 1 F 1.58 max 207604969.38 F

The Challenges of Performance Analysis

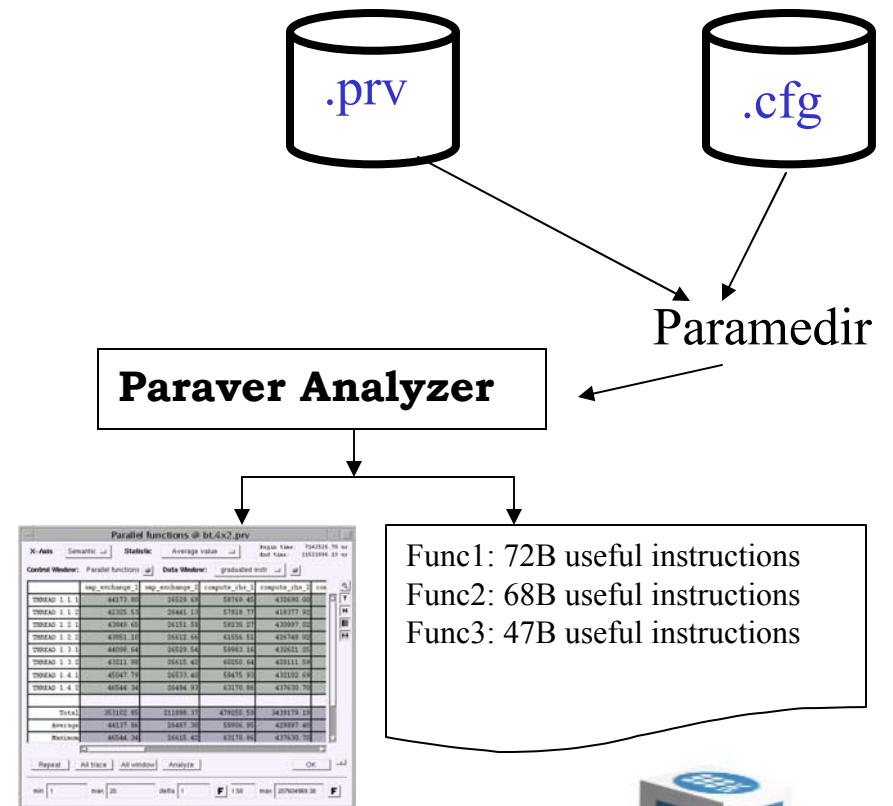
- **Gathering meaningful profiling information**
- **Interpreting performance trace:**
 - ▶ Filtering of large amounts of data
 - ▶ Calculating meaningful statistics
 - ▶ Defining new relevant metrics
 - ▶ Correlating different sets of information
- **The Expert Analyst:**
 - ▶ Repetitive visual inspection of trace data
 - ▶ Repetitive calculation and correlation of established metrics
- **The Novice User:**
 - ▶ Does not know what to look for in trace data
- **Can we automate the process?**



Paramedir

- **Command line interface to Paraver Analysis Module**
- **Input:**
 - ▶ Paraver trace file
 - ▶ Paraver configuration file
- **Output:**
 - ▶ ASCII file containing statistics
- **Configuration files pre-defined via Paraver GUI**

Paraver trace file Paraver configuration file



The Facts: Dynamic and Static Information

- **Routine and loop profile**
- **Sequential section**
 - ▶ Master thread outside of parallel region
- **Useful time**
 - ▶ Time running user code
- **Workload balance**
 - ▶ Instructions during useful time
- **Granularity**
 - ▶ Average duration of a parallel work sharing chunk
- **Estimated Parallel Efficiency**
- **Loop information**
 - ▶ Can the loop be parallelized?
 - ▶ Should a different loop be parallelized?

Where is the time spent?

Is the time spent efficiently?

Obstacles to efficiency



The ParaMed Expert System

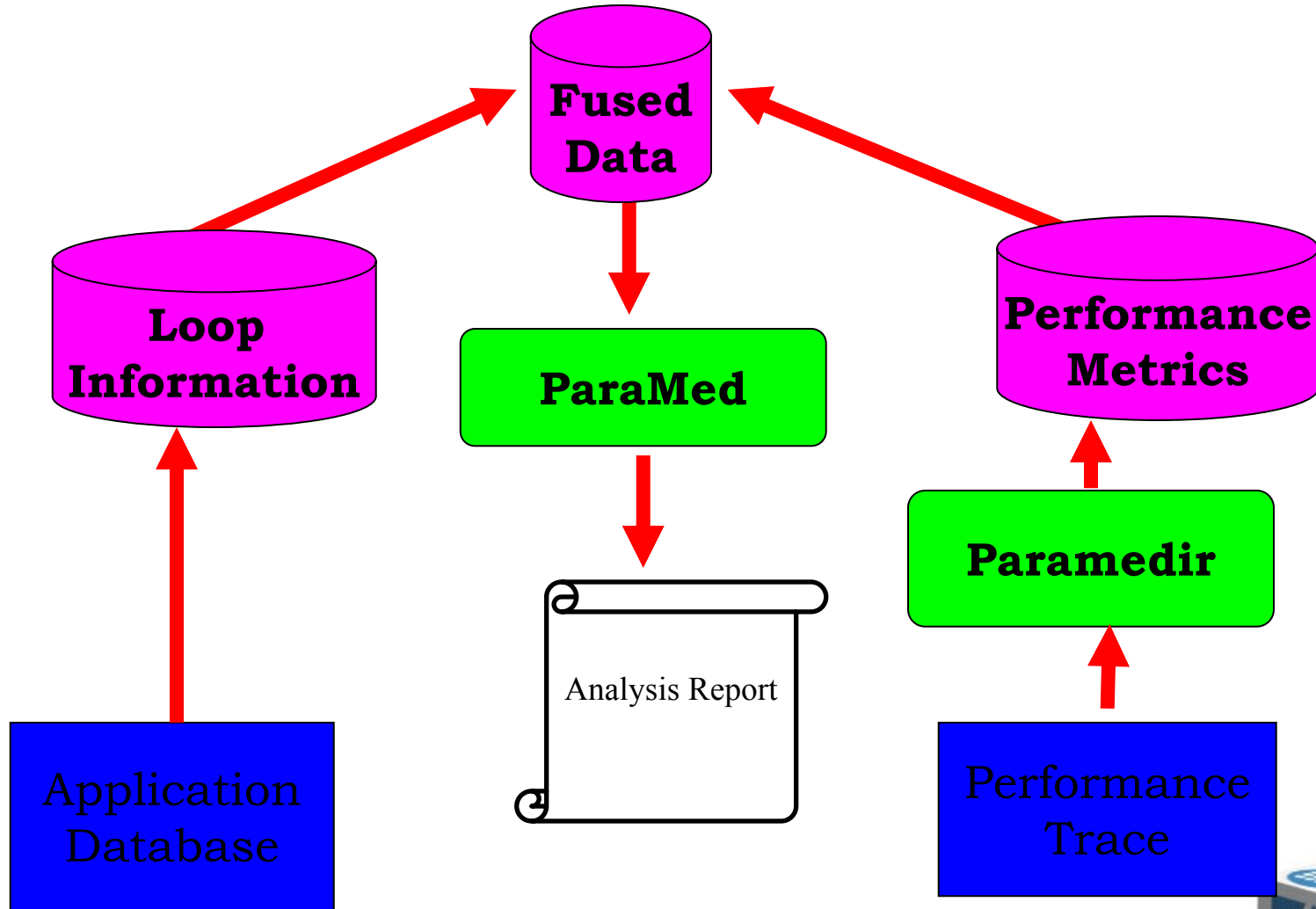
- **Implemented in CLIPS**
 - ▶ Scripting language developed at NASA
- **Assert a set of facts:**
 - ▶ Compare metrics against empirically determined threshold values
- **Apply a set of rules to the facts to derive conclusions**

IF
Time > T_Threshold
AND
Efficiency < E_Threshold
AND
Sequential > S_Threshold
AND
Granularity < G_Threshold
AND
Enclosed_in_loop

THEN
Code segment is time consuming and inefficient.
Reason: There are large sequential sections. There is inefficient fine grained parallelization.
Suggestion: Try moving parallelization an outer level.



Tool Integration



Example: Imbalanced Workload

- **PSAS Conjugate Gradient Solver**

- ▶ Component of the Goddard EOS (Earth Observing Systems) Data Assimilation System
- ▶ Nested conjugate gradient solver implemented in Fortran 90
- ▶ Time consuming routines:
 - Second level conjugate gradient solver (conjg_2)
 - Symmetric complex matrix-vector multiply (sym_Cxpy)
- ▶ The test run is performed on 4 threads



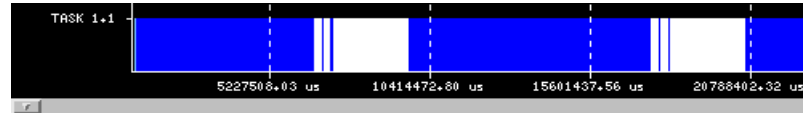
Paraver Trace File Views

Task level routine view

Routine profile:

Conjgrad: 69 %

sym_Cxpy: 26%



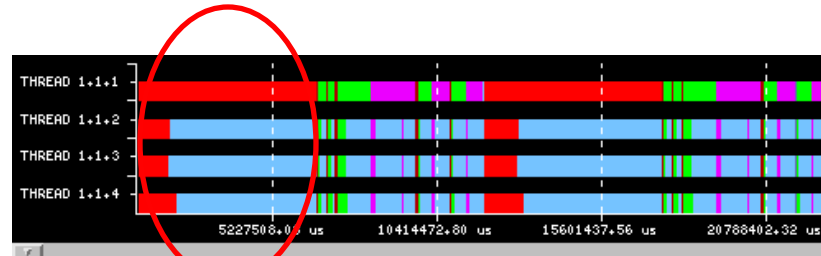
Thread level view of parallel time:

Light Blue = idle

Parallel loop in conjgrad

Parallel loop1 sym_Cxpy

Parallel loop2 sym_Cxpy



**Sequential section low,
but parallel efficiency also low.**

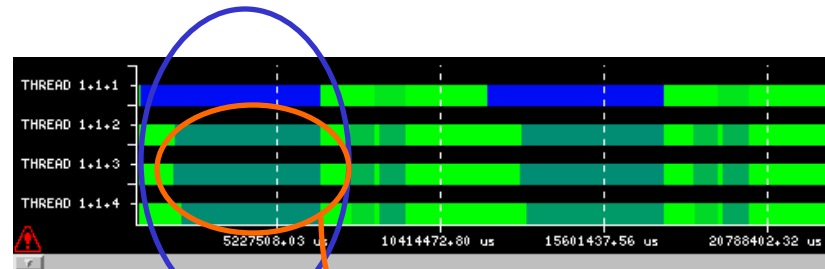
**Thread level view of # instructions
between events:**

Color gradient scheme

Dark Blue ≈ 2 Billion

Medium Blue ≈ 0.8 Billion

Light Green ≈ 0.4 Billion



**Instruction count within
within parallels is high for
thread 1, low for threads 2-4**

**Instructions outside
of parallels, while
waiting for work**



Paraver Analysis: Workload Imbalance

- Count the instructions only during useful time within parallels
- Analysis performed by Paramedir in batch mode
- Results are interpreted by ParaMed

NOTE: It is important to exclude the instructions issued during idle time!

X-Axis: Semantic Statistic Average value != 0 Begin time: 654870.39 End time: 22811318.58

Control Window: Parallel functions Data Window: graduated instr

	__mpdo_conjgr2_1.1	__mpregion_sym_expy_1.1	__mpdo_sym_expy_1.1
THREAD 1.1.1	2,227,791,591.99	380,962,039.68	291,627,231.45
THREAD 1.1.2	392,416,165.63	59,410,595.68	59,793,977.56
THREAD 1.1.3	356,659,942.37	50,060,021.52	55,138,671.39
THREAD 1.1.4	448,482,549.03	47,811,650.27	69,217,981.03
Total	3,425,350,249.02	538,244,307.16	475,777,861.42
Average	856,337,562.25	134,561,076.79	118,944,465.36
Maximum	2,227,791,591.99	380,962,039.68	291,627,231.45
Minimum	356,659,942.37	47,811,650.27	55,138,671.39
stdev	792,485,434.18	142,326,140.14	99,827,376.63
C.V.	0.93	1.06	0.84

$instr_cv > variation_threshold$



ParaMed Analysis Results

Loop List

```
sym_Cxpy:4/1/222: do jban
```

Routine List

```
conjgr2  
sym_Cxpy
```

Dismiss Help

====> Loop sym_Cxpy Loop Number 4

**** Takes 75.75 % of the execution time.
**** Runs with 46.0 % efficiency on 4 threads.
**** The loop contains parallelized loops which show load imbalance.
**** The load imbalance due to imbalance in instructions.
====>> SUGGESTED ACTION:
====>> Check the parallelized loops enclosed in loop.
====>> If number of iterations is smaller than number of threads:
====>>> Use fewer threads.
====>> If number of iterations larger than the number of threads:
====>>> Consider adding the SCHEDULE(DYNAMIC) clause to the OMP DO directive for better work load distribution

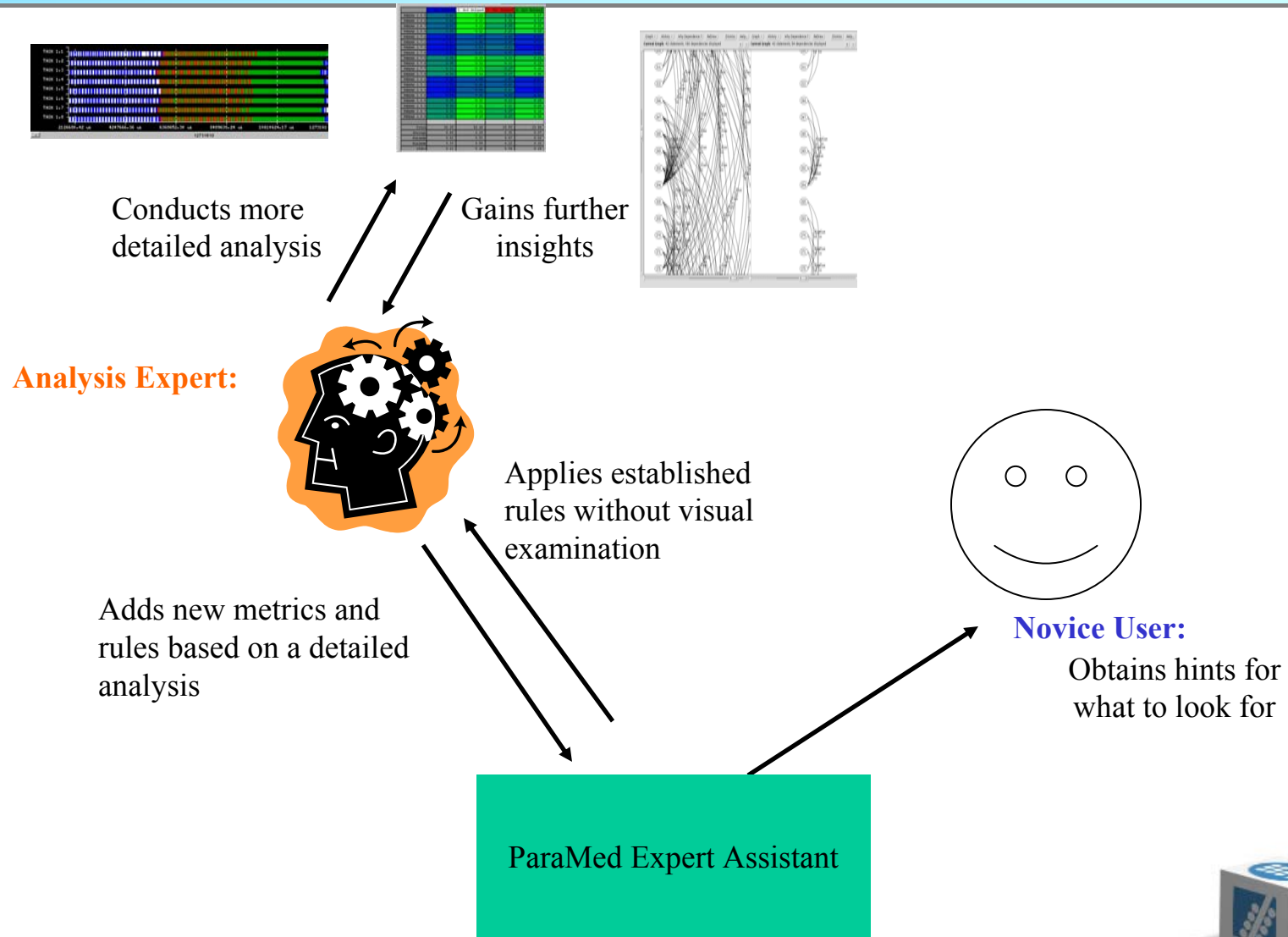
```
IF  
    Time > T_Threshold  
AND  
    Efficiency < E_Threshold  
AND  
    Sequential < S_Threshold  
AND  
    Granularity > G_Threshold  
AND  
    CV_Time > CVT_Threshold  
AND  
    CV_Instr > CVI_Threshold  
AND  
    Contains_Parallel_Loops
```

Conclusions

- **We have implemented the prototype of a parallelization expert assistant which:**
 - ▶ interfaces automatic parallelization and performance analysis
 - ▶ saves time for the expert user by automating preliminary analysis steps
 - ▶ helps the novice user to determine “what to look for”
- **A high degree of flexibility is necessary**
 - ▶ Collecting profiling information
 - ▶ Calculation of metrics
 - ▶ Adding new metrics and rules



Refining the Parallelization Process



Related Work

- **ParaWise (aka CAPTools):**
 - ▶ Developed at the University of Greenwich
 - ▶ CAPO uses CAPTools dependence analysis module
- **SUIF Explorer**
 - ▶ Developed at Stanford University
 - ▶ Interactive parallelization tool
- **KOJAK:**
 - ▶ Research Center Juelich
 - ▶ Development of an automatic performance analysis environment
- **Pardyn:**
 - ▶ University of Madison
 - ▶ Automatic detection of performance bottlenecks
- **CAPO and Paraver provide the highest level of flexibility**
 - ▶ Browsing program structure
 - ▶ Browsing performance trace

