Anfängerpraktikum

von Steffen Janz

TAU

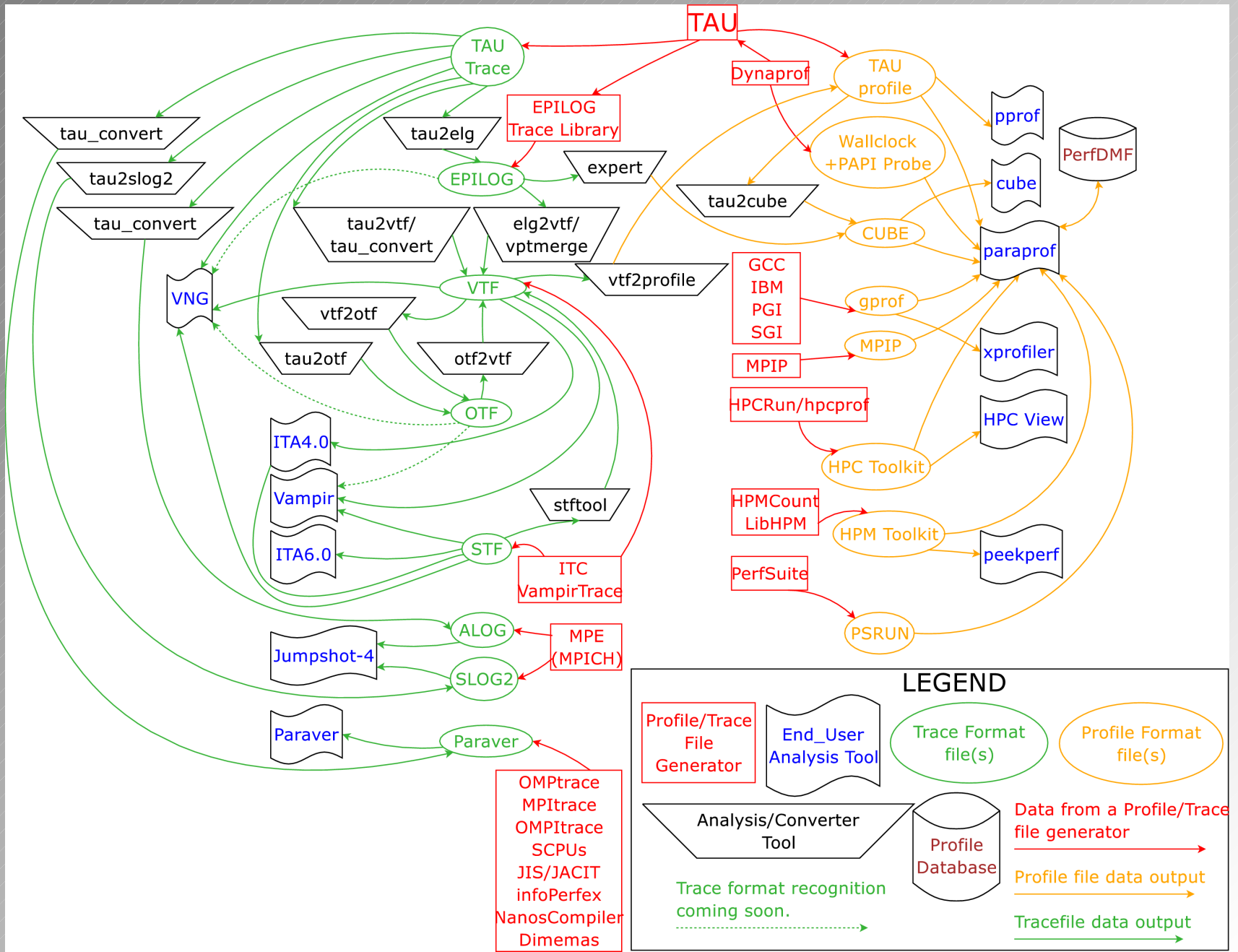
&

OTF

Inhaltsverzeichnis

Inhaltsverzeichnis (2)

Übersicht über Traceformate



TAU (Tuning and Analysis Utilities)

- (Tuning and Analysis Utilities) ist eine Sammlung von Werkzeugen zum Analysieren der Performance von C-, C++-, Fortran- und Java-Programmen
- Von PRL (Performace Research Lab)
- kann PDT (Program Database Toolkit) nutzen (empfohlen)
- Zum Einsatzgebiet des Programms gehört vor allem das Erstellen von Dokumentationen, da TAU unter anderem z.B. Klassenabhängigkeiten, Funktionsaufrufe und Template-Initialisierungen mitspeichern kann

TAU - Features

Multi-Level Performance Instrument zur Analyse

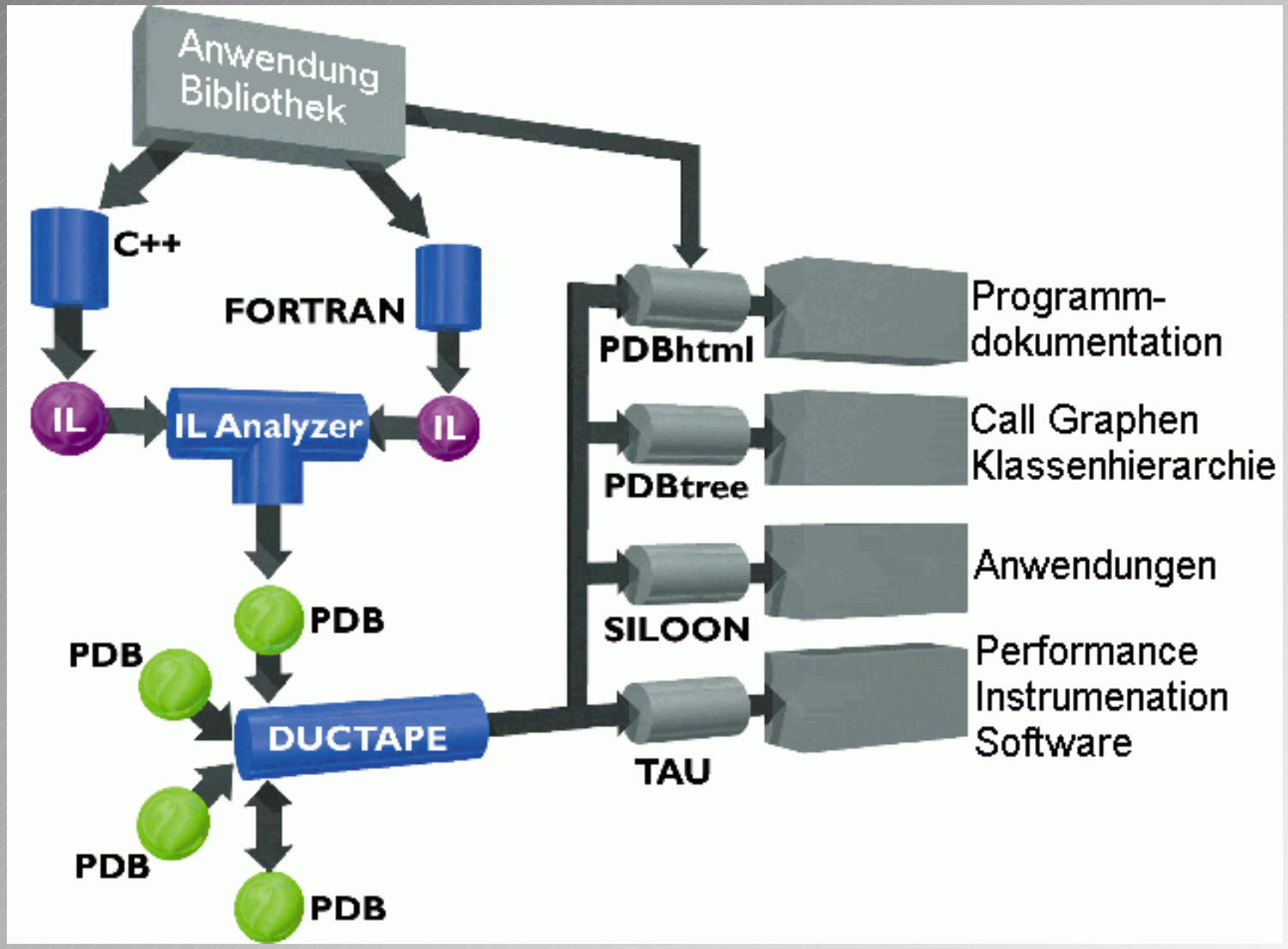
- Flexible und konfigurierbare Performanceerfassung und Kontrolle
- Parallel verteiltes und breites Analysetool
  - unabhängig von Systemarchitektur und Betriebsystem
- MultiThreding und Nachrichtenverlaufskontrolle
- Unterstützt parallele Programmierung
- Integration auch in komplexe Software möglich

TAU - Performance System Architektur

-Schritte:

- Quellcode [unser C,C++ Code]
- Objekt Code
- Bibliothek [für das TAU-Tracen]
- Binärcode
- Ausführung erzeugt die Traces (bzw. Profile)

→ Danach Analyse


Anwendung
Bibliothek
C++
FORTRAN
IL
IL Analyzer
IL
PDB
PDB
DUCTAPE
PDB
PDB
PDBhtml
PDBtree
SILOON
TAU
Programm-
dokumentation
Call Graphen
Klassenhierarchie
Anwendungen
Performance
Instrumenation
Software

PDT (Program Database Toolkit)

- PDT ist ein Framework zur Analyse von Codes in jeder Sprache, unter anderem in C/C++
- Ermöglicht automatische Instrumentierung des Codes
- Erzeugt die automatische Dokumentation (Tracen) von Klassenhierarchien, Funktionsaufrufe und Graphen
- Analysiert statisch und dynamisch
- Stellt eine Bibliothek bereit zur besseren Performanceanalyse

TAU - Benutznugshinweis

- In allen Beispielen wurde C verwendet, jedoch funktioniert TAU natürlich auch mit C++ (Kompiler: tau_cxx.sh) oder F90 (Fortrankompiler au_f90.sh)
- Ich zeige die Anwendung anhand meines Beispiels 3 gewinnt (oder auch bekannt als XXO)

TAU - Benutzung mit Automatischer Instrumentierung

- Zunächst einmal bracht man eine C, C++ Quelldatei, die wir jetzt kompilieren (aber nicht mit g++)
- tau_cc.sh Name_der_Datei -I *1/include -L *2/lib -TracerArt*3 -o kompilierte_Datei_Name
- Hinweise fürs Ersetzen:

  *1, *2 müssen auf die korrekten Pfader verweisen:

  *1 -> VERZEICHNIS/TAU/install/tau-otf/include/

  *2 -> VERZEICHNIS/TAU/install/tau-otf/i386_linux/lib/

  TracerArt*3 wählt die Tracereinstellungen

In meinem Beispiel:

tau_cc.sh 3w.cpp -I /home/sjanz/include/ -L /home/sjanz/lib -lTAU_traceinput-mpi-pthread-pdt-trace-mpitrace -o 3w

TAU - Benutzung mit Automatischer Instrumentierung (2)

- Hinweis:
  - Fehler im C/C++ Code werden dann angezeigt, aber die Zeilenangabe kann falsch sein!
- Wenn alles geklappt hat, dann sollte jetzt im gewählten Verzeichnis eine neue Datei vorhanden sein:
- kompilierte_Datei_Name (den wir uns selbst gewählt haben)
  - z.B. 3w

TAU - Benutzung mit Automatischer Instrumentierung (3)

- Als nächstes Programm ausführen / laufen lassen:
  - ./3w
- Nach dem Ausführen (und Beenden) des Programms haben wir nun 2 weitere Dateien:
  - events.0.edf
  - tautrace.0.0.0.trc
- Beide zusammen enthalten die Tracespur für TAU!
- Sie wurden automatisch erzeugt, als wir das Programm ausgeführt haben

TAU - Benutzung mit Automatischer Instrumentierung (4)

- Die beiden Dateien (events.0.edf, tautrace.0.0.0.trc) enthalten die automatisch von PDT erzeugten Logeinträge.
- Sie enthalten:
  - Funktionsaufrufe (Methoden, Prozeduren) & deren beenden
  - Zeitmessung (als Zeitpunkt der einzelnen Events)
  - Call Tree
  - Threadnummer
  - Nodenummer

Anschauen der Traces – Dump (1)

- Es gibt weitere (und bessere) Tools zum betrachten von Tracefiles wie z.B. Vampire (oder für Profile Paraprof), doch schauen wir es und mal nur per –dump befehl an:
  - /PFAD_DAHIN/tau_convert -dump tautrace.0.0.0.trc events.0.edf

  In meinem Beispiel:

  /home/jkunkel2/TAU/install/tau-otf-no-mpi/i386_linux/bin/tau_convert -dump tautrace.0.0.0.trc events.0.edf

mc - master1:~/tester/ohne

```
#  creation program: tau_convert -dump
#     creation date: Sep-27-2007
#    number records: 98
# number processors: 1
# max processor num: 0
#   first timestamp: 1190900197437940
#    last timestamp: 1190900219230929

#=NO= ========================EVENT== ==TIME [us]= =NODE= =THRD= ==PARAMETER=
    1                        "EV_INIT" 1190900197437940      0      0             3
    2 "int main() [{3w.cpp} (75,1)-( 1190900197437940      0      0             1
    3                     "WALL_CLOCK" 1190900197438115      0      0   1190900197
    4 "void warte(long) [{3w.cpp} (2 1190900197438232      0      0             1
    5 "void warte(long) [{3w.cpp} (2 1190900197498867      0      0            -1
    6 "void warte(long) [{3w.cpp} (2 1190900197498876      0      0             1
    7 "void warte(long) [{3w.cpp} (2 1190900197568948      0      0            -1
    8 "void warte(long) [{3w.cpp} (2 1190900197568953      0      0             1
    9 "void warte(long) [{3w.cpp} (2 1190900197630937      0      0            -1
   10 "void warte(long) [{3w.cpp} (2 1190900197630941      0      0             1
   11 "void warte(long) [{3w.cpp} (2 1190900197693616      0      0            -1
   12 "void warte(long) [{3w.cpp} (2 1190900197693625      0      0             1
   13 "void warte(long) [{3w.cpp} (2 1190900197755829      0      0            -1
   14 "void warte(long) [{3w.cpp} (2 1190900197755833      0      0             1
   15 "void warte(long) [{3w.cpp} (2 1190900197834494      0      0            -1
```

Anschauen der Traces – Dump (2)

#=NO=	=EVENT	TIME [us]	=NODE=	=THRD=	PARA-METER
Ereignis-nummer (durch-zählend)	Beschreib-ung	Zeitpunkt in µ Sekunden	Node-nummer	Thread-nummer	Rückgabe-wert

- **Ereignisnummer:**
  - Immer aufzählend, start bei 1

Anschauen der Traces – Dump (3)

- **Event:**
  - Kurze Beschreibung was aufgerufen wurde (siehe weiter unten) bei Funktions-/Methodenaufrufen wie im C Code angegeben mit Parametertyp
  - Parameterwert wird NICHT gespeichert
- **Time:**
  - Genaue Zeit, an der dem dies ausgeführt worden ist
- **Node:**
  - Zeigt die Nodenummer an
- **Thread:**
  - Zeigt die Treadnummer an

Anschauen der Traces – Dump (4)

Bedeutung der Parameter

Event	Parameter	Aktion
"EV_INIT"	3	Start des Programms
"int/void/…"	1	eine Methode/Funktion wurde aufgerufen
"int/void/…"	-1	die Methode/Funktion wurde beendet
"WALL_CLOCK"	*irgendeineZahl*	Startzeit-/Endzeitpunkt

Beispiel

Beispiel an 3 gewinnt zeigen
(3w.c)

Bewertung der automatisch erzeugten Traces

- Es werden alle Start- & Stoppzeitpunkte genau gespeichert, von
  - Funktionsaufrufen
  - Methoden
  - Klassen
  - gesamter Zeitbedarf
- Leider gehen alle Variablenwerte oder Benutzereingaben verloren und wir haben keine „Zwischenzeiten“ innerhalb einer Routine / Funktion

Benutzerevents zusätzliche Informationen einbauen

- Was sind Benutzerevents
  - Manuelles hinzufügen von zusätzlichen Informationen durch den Benutzer in die TAU Traces (in den Quellcode)
- Wozu
  - Speicherung von Variablenwerten, Benutzereingaben oder anderen Informationen während der Laufzeit
- Wie
  - Diese Makros einzubauen erfordert jedoch etwas mehr Aufwand, da diese von Hand manuell eingefügt werden müssen
  - Alternativ eigene Traces schreiben

Benutzerevents (2)

- Um die zusätzlichen Informationen speichern zu können benötigen wir 2 weitere Dateien (im weiteren tau.edf und out.odf genannt). Diese werten direkt (vom Benutzer) in unserem Programmcode an entsprechender Stelle eingefügt.
- Diese zusätzlichen Informationen werden nur in den beiden Dateien gespeichert, sie beeinflussen die normale Tracefile nicht!
- Zunächst einmal brauchen wir die Bibliothek #include <TAU_tf.h>

Benutzerevents (3)

Wie benötigen weiterhin:

- Filehandler (am Anfang unseres Codes!)
  - Ttf_FileHandleT ausgDatei;
    // [Filehandler bzw. Konstrukor, "ausgDatei" ist Beispielname]
- Ausgabedateien
  - ausgDatei = Ttf_OpenFileForOutput("tau.trc","tau.edf");
    // [legt die Ausgabedateien fest]
    // [tau.trc, tau.edf von mir gewählte Namen!]
- Speichern unserer Daten als Trace mittels
  - Ttf_CloseOutputFile(ausgDatei);
    // [Dateien speichern und schließen]

Benutzerevents (4)

- Die Funktion, die tf_* Tracen soll, muss folgendes Format haben (hier Beispiel mit main):

```
int main(int argc, char** argv) {
        Code ...
}
```

- Zur Sicherheit empfiehlt es sich noch eine Abfrage zu machen:

```
if (ausgDatei == NULL) {
        fprintf (stderr, "tau.trc oder tau.edf kann nicht angelegtwerden!\n");
        return -1;
}
```

Benutzerevents (5)

```
int main(int argc, char** argv) {
    // erstelle den File-Handler [Konstrukor]
    Ttf_FileHandleT file;
    // öffne die Ausgabedateien
    file = Ttf_OpenFileForOutput("tau.trc","tau.edf");
    // Dateien nicht vorhanden -> Fehler
    if (file == NULL) {
        fprintf (stderr, "Error openging trace for output\n");
        return -1;
     ...
     // Unser Programm // Code //
     ...
     Ttf_CloseOutputFile(file);
     return 0;
    }
```

Benutzerevents – mögliche Befehle

- int Ttf_DefClkPeriod(ausgDatei, Zeitperiode);
- int Ttf_EndTrace(Benutzerdaten, Nodenummer, Threadnummer);
- int Ttf_DefThread(Benutzerdaten, Nodenummer, Threadnummer, Name);
- int Ttf_DefStateGroup(ausgDatei, Gruppennummer, Gruppenname);
- int Ttf_DefState(ausgDatei, Tokennummer, Name, Gruppennummer);
- int Ttf_DefUserEvent(ausgDatei, Benutzereventnummer, Benutzereventname, Autozähler);
- int Ttf_EnterState(ausgDatei, Zeit, Nodenummer, Threadnummer, Tokennummer);
- int Ttf_LeaveState(ausgDatei, Zeit, Nodenummer, Threadnummer, Tokennummer);
- int Ttf_SendMessage(ausgDatei, Zeit, VonNodenummer, VonThreadnummer, ZielNodenummmer, ZielThreadnummer, Nachrichtengröße, int NachrichtenTag);
- int Ttf_RecvMessage(ausgDatei, Zeit, VonNodenummer, VonThreadnummer, ZielNodenummmer, ZielThreadnummer, Nachrichtengröße, int NachrichtenTag);
- int Ttf_EventTrigger(ausgDatei, Zeit, Nodenummer, Threadnummer, Benutzereventnummer, BenutzereventWert);

Weitere TAU Möglichkeiten

- TAU kann auch den verfügbaren Arbeitsspeicher überwachen

  (Standardeinstellung alle 10 Sekunden, wenn aktiviert):

  - TAU_TRACK_MEMORY(); //Initiiert
    - TAU_ENABLE_TRACKING_MEMORY(); //Startet
    - TAU_DISABLE_TRACKING_MEMORY(); //Hält an
- Ebenso können auch Memoryleaks gesucht werden:
  - -optDetectMemoryLeaks beim kompilieren

Weitere TAU Möglichkeiten (2)

- Speichern von Variablenwerten
  - int Ttf_EventTrigger(ausgDatei, Zeit, Nodenummer, Threadnummer, Benutzereventnummer, BenutzereventWert);
- Anderen Tools wie:
  - TDL/POET
    - ein Tool für Events aus Datenzugriffen

Beispiel (2)

Beispiel an 3 gewinnt zeigen
(4w.c)

Hinweise zu benutzerspezifischen Events

- Alle unsigned int lassen sich auch mit negativen Zahlen im Code festlegen, diese aber werden dann mir einem "Warning" als Ausgabe in unsigned umgewandelt (Betrag)
- Namen müssen nicht verschieden sein, bei Mehrfachinitialisierung zählt die letzte
- Alle Namen sind Case-Sensitive [Groß- und Kleinschreibung!]

Hinweise zu benutzerspezifischen Events (2)

- Mehrfaches betreten des gleichen Statements hat keine Auswirkung
- Anzahl Leavestates <= Anzahl Enterstates sonst Fehler
- ABAB wird zu ABA, nur bei verschiedenen Threads wird B nicht mitbeendet
- Threadnamen oder Nodenamen werden nicht gespeichert

Hinweise zu benutzerspezifischen Events (3)

- Bei einem Programmabsturz oder Fehler bleiben die Tracefiles events.0.edf, tautrace.0.0.0.trc erhalten und werden automatisch rausgeschrieben mit allen Ereignissen bis zu diesem Fehler!
- Hingegen bei tau.edf, tau.trc sind mit einem Programmabsturz/Fehler alle Logdateiinhalte verloren, da die beiden Dateien zwar erzeugt werden (und damit leer sind), aber die geloggten Einträge nicht gespeichert wurden.
- Wird Ttf_CloseOutputFile(ausgDatei); nicht erreicht, sind ebenfalls alle Tracedaten von tau.edf, tau.trc verloren!

Zeitliche Erfassungsmöglichkeiten bei TAU

- Static timers (statische Zeiterfassung)
- Dynamic timers (dynamische Zeiterfassung)
- Static phases (Messung ganzer Routinen)
- Dynamic phases (Messung von einzelnen Routinendurchläufen)

Static Timer (statische Zeiterfassung)

- Standardzeitmessung (fester Start und Endpunkt im Code):
  - Festgelegter Name
  - Gruppe (die den Timer startet)
  - Benötigte Zeit vom Start bis zum Endpunkt
    - z.B. Start und Ende des Prozesses

-> Geeignet für die Erfassung von bestimmten

(Zeit-)Punkten während der Ausführung

Dynamic Timer (dynamische Zeiterfassung)

- Ähnelt stark dem Static Timer, jedoch kann der Name mit Hilfe eines Konstruktors im Code verändert werden! Gespeichert wird:
  - Name (auch durch Konstruktor festlegbar)
  - Gruppe (die den Timer startet)
  - Benötigte Zeit vom Start bis zum Endpunkt
    - z.B. Start und Ende des Prozesses

-> Geeignet um einzelne Iterationen zu trennen und zu analysieren

Static Phases (Messung ganzer Routinen)

- Eine Anwendung kann in mehrere Laufphasen unterteilt werden. Eine Phase (bzw. z.B. eine Funktion) hat einen eindeutigen Start-Ausdruck (oder Punkt), dessen Laufzeit sich natürlich auch durch weitere Aufrufe (direkte und indirekte) in die Länge ziehen kann:
  - Zeitverbrauch für eine Phase/Routine (alle Laufzeiten der Phase werden addiert)
  - Speichern des aktuellen Ausdrucks
  - Sowohl direkte als auch indirekte Aufrufe
  - Start und Ende jeder Phase

-> Geeignet um den Gesamtaufwand von Unterroutinen zu erfassen

Dynamic Phases (Messung einzelner Routinendurchläufe)

- Ähnelt sehr den Static Phases, jedoch kann auch hier, wie bei den Dynamic Timer die einzelnen Namen per Konstruktor verändert werden. Vorteil: Ein eindeutiger "neuer" Name für jede einzelne Iteration. Getraced wird:
  - Zeitverbrauch für eine Phase/Routine (alle Laufzeiten der Phase werden addiert)
  - Speichern des aktuellen Ausdrucks
  - Sowohl direkte als auch indirekte Aufrufe
  - Start und Ende jeder Phase

-> Geeignet um einzelne Routinendurchläufe zu analysieren (z.B. beim Durchlauf Nr. 4)

-> Herausfinden von schlechten Iterationsroutinen

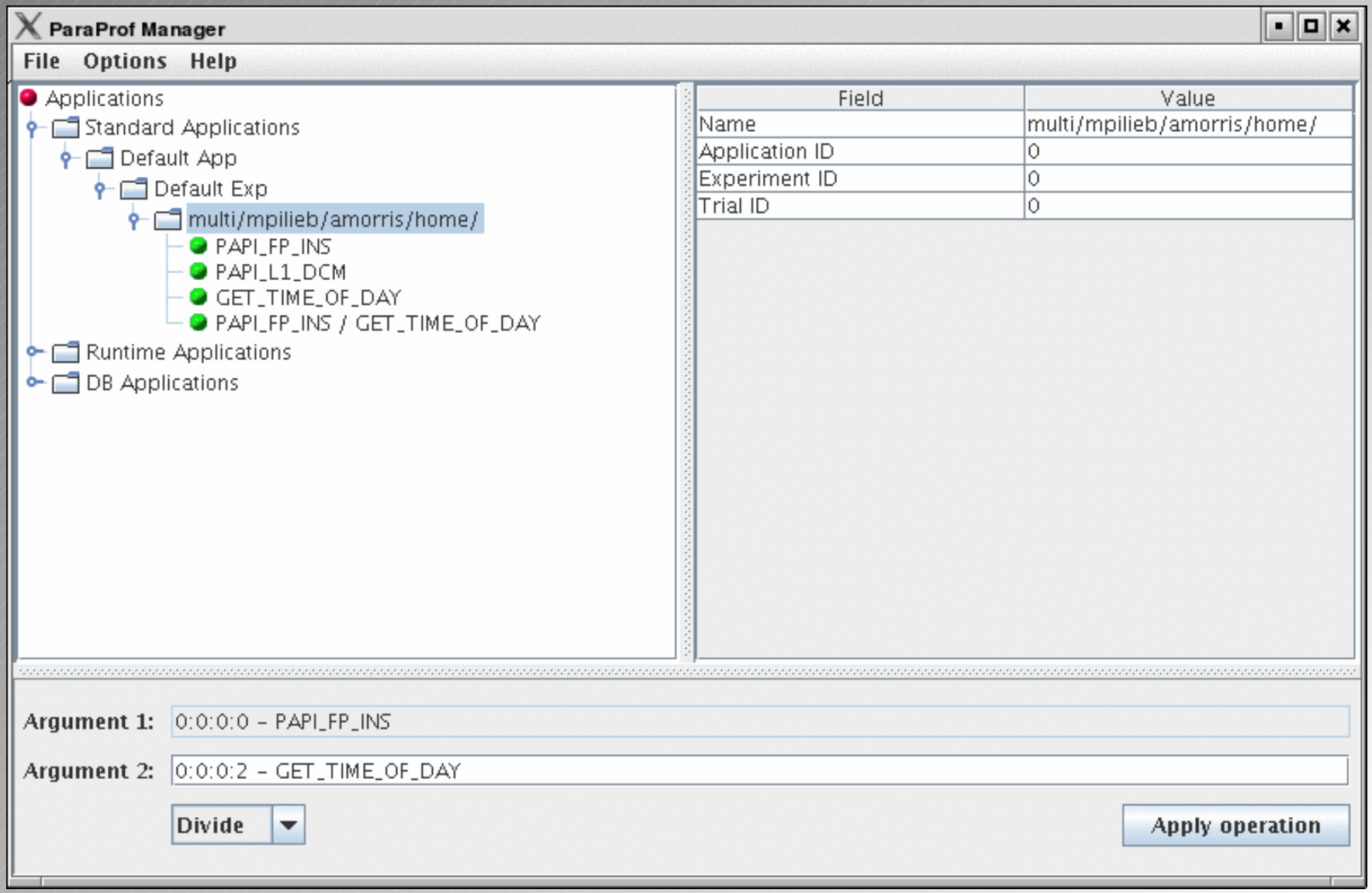
Callpaths

- In Phasenprofile erkennt man die Beziehung zwischen den Routinen (bzw. Funktionen) und deren vorangegangener Aufruf oftmals nur schwer. Daher gibt es Callpath, der alle Aufrufe als graphischen Baum darstellen kann. Callpath enthält:
- Eine Liste aller Routinen und deren weiteren Aufrufe
- Stufenweise, wie ein Baum
- Maximale mitzutracende Tiefe einstellbar

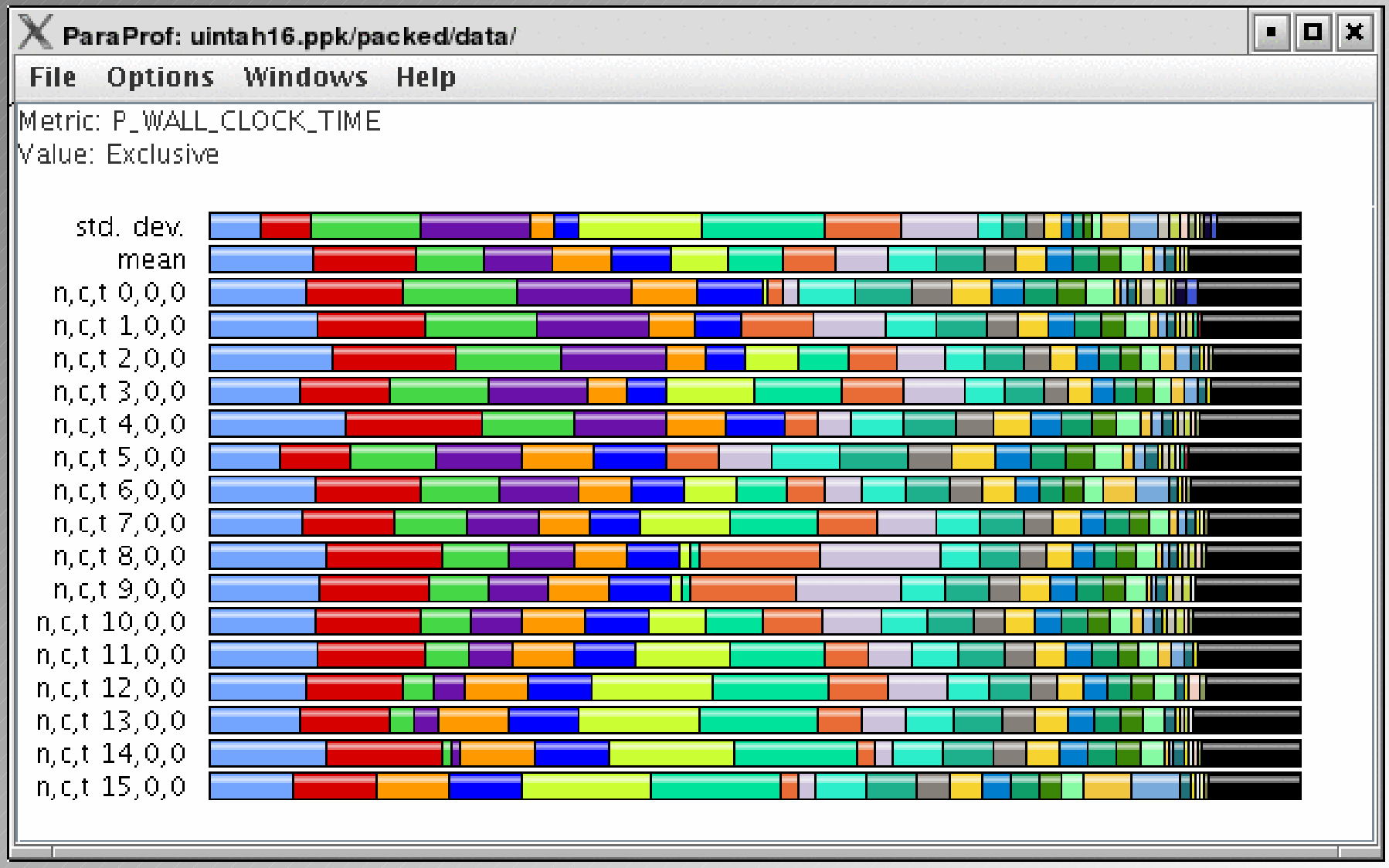
Tools – Paraprof

- Anzeigen von Profiltraces zur genauen Analyse
- Profile sind (z.B. Node) gebunde Tracefiles zu einzelne Analyse
- Ermöglicht das Erstellen von Statistiken (mit Hilfe einer selbstdefinierbaren Metrik)
- Gute Visualisierungsmöglichkeit

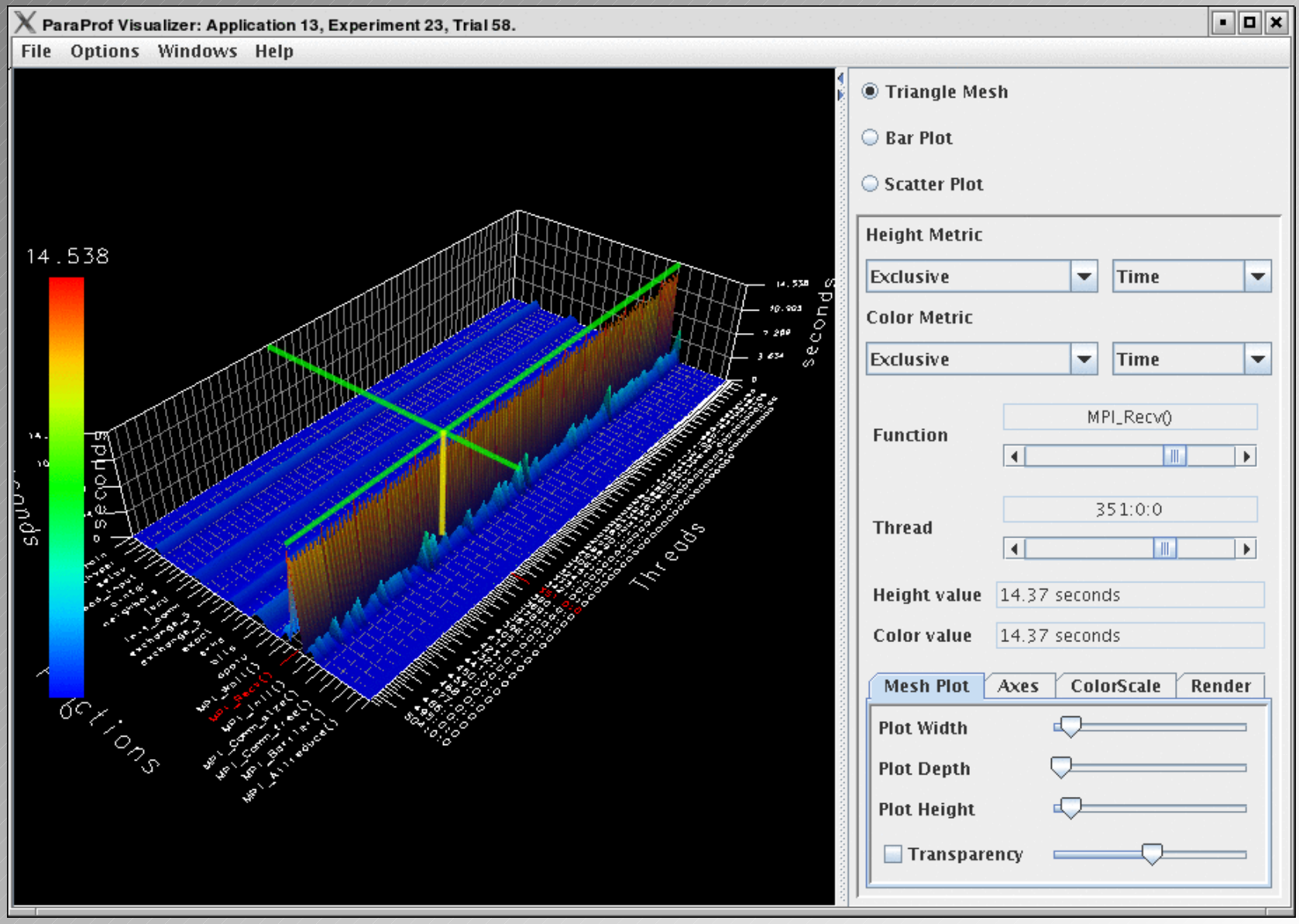
Paraprof – Metrik Einstellungen



Paraprof – Standardansicht



Paraprof – 3D Visualisierung



Paraprof (2)

- Bis hin ins Detail:
  - Wann?
    - Wie oft?
  - Ruft etwas anderes auf?
    - Übergebene Parameter?*
  - benötigte Zeit?
    - Einzelne Zeitpunkte zu beliebigen Ereignissen*
  - Dafür verwendeter Speicherplatz

*[bei entsprechend eingefügter Benutzerevents]

Paraprof (3)

- Darstellung der ganzen Performance als
  - Reine Zahlen
  - Tabellen / Balken
  - Graphische 3D Darstellung
  - Baumdarstellung (Callpath)

  Zur genauen Analyse von Programmschwächen

Unterschiede in C und C++

- Kompiler
  - tau_cxx.sh für C++
  - tau_cc.sh für C
- Benutzerevents
  - C++
    - TAU_PROFILE // erzeugt aus einer C++ Funktion ein Profil
    - TAU_PHASE // erstellt ein C++ Profil von einer Phase
  - C
    - TAU_PROFILE_DECLARE_TIMER // deklariert einen Timer
    - TAU_PROFILE_CREATE_TIMER // erzeugt einen Timer

OTF (Open Trace Format)

- Alternative zu TAU
- Entwickelt von der Technischen Universität Dresden
- Lässt sich aus TAU konvertieren
  - PFADDAHIN/tau2otf tau.trc tau.edf NAME.otf

OTF - Features

- Plattform unabhängig
- Zugriff möglich auf
  - Prozesse
  - Zeitintervalle
- API / Interface & Analyse
  - Lesen und schreiben von mehreren Dateien gleichzeitig
  - Filtersysteme (Nur interessante Routinen Tracen)
  - Unterstützt parallele Systeme (aufteilen der Daten in Streams)
- Aufteilung der Traces in Streams, die bei Bedarf wieder zur Masterfile vereinigt werden können

OTF – Traces

Enthält:

- Zeitpunkten von Events
  - Mit dem Namen der jeweiligen Prozessgruppe
- Statements
  - Enter States / Leave States (vom Benutzer manuell hinzuzufügen, wie in TAU)
  - Häufigkeitenzählung der States
- Verteilt auf viele einzelne Streams
  - Bei Bedarf zu einer Master Datei vereinigt zur statistischen Analyse
- Snapshots
  - Momentanaufnahmen (z.B. vom benötigten Arbeitsspeicher)

OTF – Traces (2)

- OTF nutzt eine spezielle ASCII Kodierung (Präfixkodierung) um
  - Festplattenspeicher zu sparen
  - Schnelle Lese-/Such-/Schreibzugriffe zu ermöglichen
  - Plattformunabhängige Datendekodierung

Streams

- OTF bietet an, die Tracedaten in verschiedene Streams (einzelne Tracedateien) zu schreiben
  - Jeder Stream kann aus mehreren Dateien bestehen, mit den möglichen Inhalten:
    - Funktionsaufrufen
    - Zeitpunkten
    - Statements (Leave / Enter)
    - Statusinformation / Hardwaredaten (Snapshot)
    - Nachrichteneingänge / -ausgänge
    - Statistischen Informationen
  - Meistens werden diese jedoch in getrennten Dateien gespeichert

Streams (2)

Verschiedene Dateien:

- Masterdatei NAME.otf *1
- Globale Definitionen: NAME.0.def *1
- Lokale Definitionen: Name.x.defs *2
- Events: NAME.x.events *1 *2
- Snapshots: NAME.x.snaps [optional] *2
- Statistische Daten: Name.x.stats [optional] *2

*1Hinweis: fehlt einer dieser Dateien sind die Traces ungültig!

*2 x wird durchgezählt!

Erzeugen von OTF Traces

- OTF Traces lassen sich am einfachsten aus TAU Traces erstellen, die man per tau2otf (ein Tool) in OTF konvertiert
- Alternative können natürlich auch OTF Traces wie TAU Benutzerevents von Hand manuell in den Code eingefügt werden

OTF im Code – Beispiel

- Kopfzeilen (ähnelt stark TAU!)
  - #include <otf.h>
  - #include <assert.h>
  - **int** main(**int** argc, **char**** argv) { ... Code ...}
- Filehandler
  - OTF_FileManager* manager;
  - OTF_Writer* writer;
- Filehandler initialisieren und beschränken auf 100 Dateien
  - manager= OTF_FileManager_open(100);
  - assert(manager);

OTF im Code – Beispiel (2)

- Initialisiert das Schreiben(writer) und öffnet die Datei „test“ (schreibt nur 1 Stream)
  - writer = OTF_Writer_open("test", 1, manager);
  - assert(writer);
- Sourcecode mit Events
  - OTF_Writer_writeDefTimerResolution(writer, 0, 1000);
  - OTF_Writer_writeDefProcess(writer, 0, 1, "TestMe1", 0);
  - OTF_Writer_writeDefFunctionGroup(writer, 0, 1000, "Alle Funktionen");
  - OTF_Writer_writeDefFunction(writer, 0, 1, "main", 1000, 0);

OTF im Code – Beispiel (3)

- Statements
  - OTF_Writer_writeEnter(writer, 10000, 1, 1, 0);
  - OTF_Writer_writeLeave(writer, 20000, 1, 1, 0);
- Abschließend (Ende des Quellcodes):
  - OTF_Writer_close(writer);
  - OTF_FileManager_close(manager);
  - **return** 0;
- }

Tools – Streams Vereinigen

- otfmerge
  - Vereinigung mehrerer Streams
  - Hinzufügen von statistischen Daten aus einer weiteren Datei
  - Hinzufügen von Momentanaufnahmen (Snapshots), bzw. weietren Daten

Tools - Profilerstellung

otfaux

- fügt zusätzliche Statistiken oder aktuelle Werte ein, die frei konfigurierbar sind (Exclisives/Inclusives/Times/…)
- Statistiken setzten sich automatisch selbst fort und erstellen Profile
- Die Orginaltrace Dateien bleiben unangetastet
- Aktualisierungsintervalle beliebig wählbar [immer, alle x …]

Unterschiede TAU und OTF

	TAU	OTF
Trace Erstellung	+ Automatische Instrumentierung + zusätzliche Events manuell hinzuzufügen – aber umständlich	- keine Automatische Tracererstellung + zusätzliche Events manuell hinzuzufügen – aber umständlich
Profilerstellung	+ möglich (automatisch)	+ möglich mit otfaux
Aufteilen in einzelne Dateien	+ möglich nur mit entsprechender Konfiguration	+ einfach möglich
Neues Erzeugen einer Masterdatei	- Nicht mehr möglich	+ möglich (otfmerge)

Unterschiede TAU und OTF (2)

	TAU	OTF
Konvertierbarkeit	+ flexibler (auch in OTF konvertierbar)	- weniger flexibel (nicht in TAU konvertierbar)
Information der Traces	+ gut (vor allem durch die Benutzerevents)	+ gibt viel mehr Möglichkeiten zum Tracen

Performance

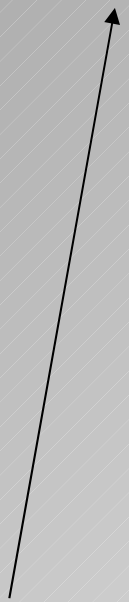
- Das Instrumentieren (mitloggen) der ganzen Events kostet sowohl Prozessorlast als auch zusätzlichen Haupt- und Festplattenspeicher
- Die zusätzliche „Last“ variiert sehr stark von der Art der Instrumentierung bzw. der Art des Programms

Performance (2)

Wie lange braucht dieser Code?

```
int plus(int a){
    a++;
    return a;
}

// Fortsetzung siehe rechts oben
// 20 000 000 mal Plus 1 Funktion
```



```
int main(){
    long i = 0;
    printf("Start\n");
    while (i < 20000000){
        i = plus(i);
        if ((i % 1000000) == 0){
            printf(".");
        }
    }
    printf("\nFertig!");
    return 0;
}
```

Performance (3)

- Ausführung ganz ohne Tracing (3 mal getestet, in Sekunden):

0m0.345 user	0m0.344 sys	0m0.000 real
0m0.307 user	0m0.304 sys	0m0.000 real
0m0.307 user	0m0.304 sys	0m0.004 real

-> Also sehr schnell!

Performance (4)

- 3 mal ausgeführt mir automatischer Instrumentierung:

1m8.941s user 0m23.013s sys 0m38.650s real

1m12.814s user 0m24.442s sys 0m36.854s real

1m13.257s user 0m24.450s sys 0m37.762s real

Eine Katastrophe! Sehr großes Overhead!

Zeiten user: ~200fach sys: ~80fach real: ~900fach

Performance (5)

- 3 mal ausgeführt mit selbst eingesetzten Benutzerevents

1m43.401s user	0m29.126s sys	0m42.203s real
1m29.967s user	0m30.082s sys	0m41.519s real
1m39.862s user	0m29.166s sys	0m42.567s real

Noch schlimmer ...

user: ~300 fach sys: ~100 fach real: über ~1000fach

Performance (6)

- Nebenbei sei erwähnt, dass dabei ein Tracefile von über 900MB geschrieben wurde ...
- Deswegen nur Benutzen wo sinnvoll und notwendig, auf keinen Fall bei großen Schleifenanzahlen!
- Im Schnitt kann man davon ausgehen, dass ein Event ca. 25 Byte Speicher braucht und bei größer Anzahl sich das sehr schnell aufsummiert!

Fragen???

Dann bitte melden und fragen!

Feedback

- Völlig unbekanntes Thema bekommen
  - Oje ...
- Lässt sich natürlich nicht auf Windows ohne weiteres zum laufen bringen
  - Oje ...
- Zunächst kein eigenes Feedback möglich
  - Ahhh! (Ist für mich sehr wichtig, zu sehen, dass ich was gemacht habe!)
- In Wikipedia nichts vernünftiges gefunden
  - Oje ...

Feedback (2)

- Jeden Tag in die Uni fahren (Fahrzeit 2 Stunden für beide Richtungen)
  - Zum Glück nicht :)
- Arbeiten per Remote
  - Funktioniert zum Glück ziemlich gut bis auf gewisse Anfangsfehler ...
- Arbeiten in der Shellbox
  - Was DOS? So etwas wird noch benutzt XD

Feedback (3)

- sjanz@master1:~$ mkdir tester

-sh: mkdir: command not found

- sjanz@master1:~$ cd tester

-sh: cd: tester: No such file or directory

- sjanz@master1:~$ dir

-sh: dir: command not found

- sjanz@master1:~$ ls

-sh: ls: command not found

→ Pfad war falsch

Feedback (4)

sjanz@master1:/$ dir

-> bin home media sbin stable.lenny var
boot initrd mnt srv sys vmlinuz.old
cdrom initrd.img.old opt stable tftpboot
dev lib proc stable.etch tmp
etc lost+found root stable.feisty usr

Ein paar Sekunden später:
sjanz@master1:/~ dir

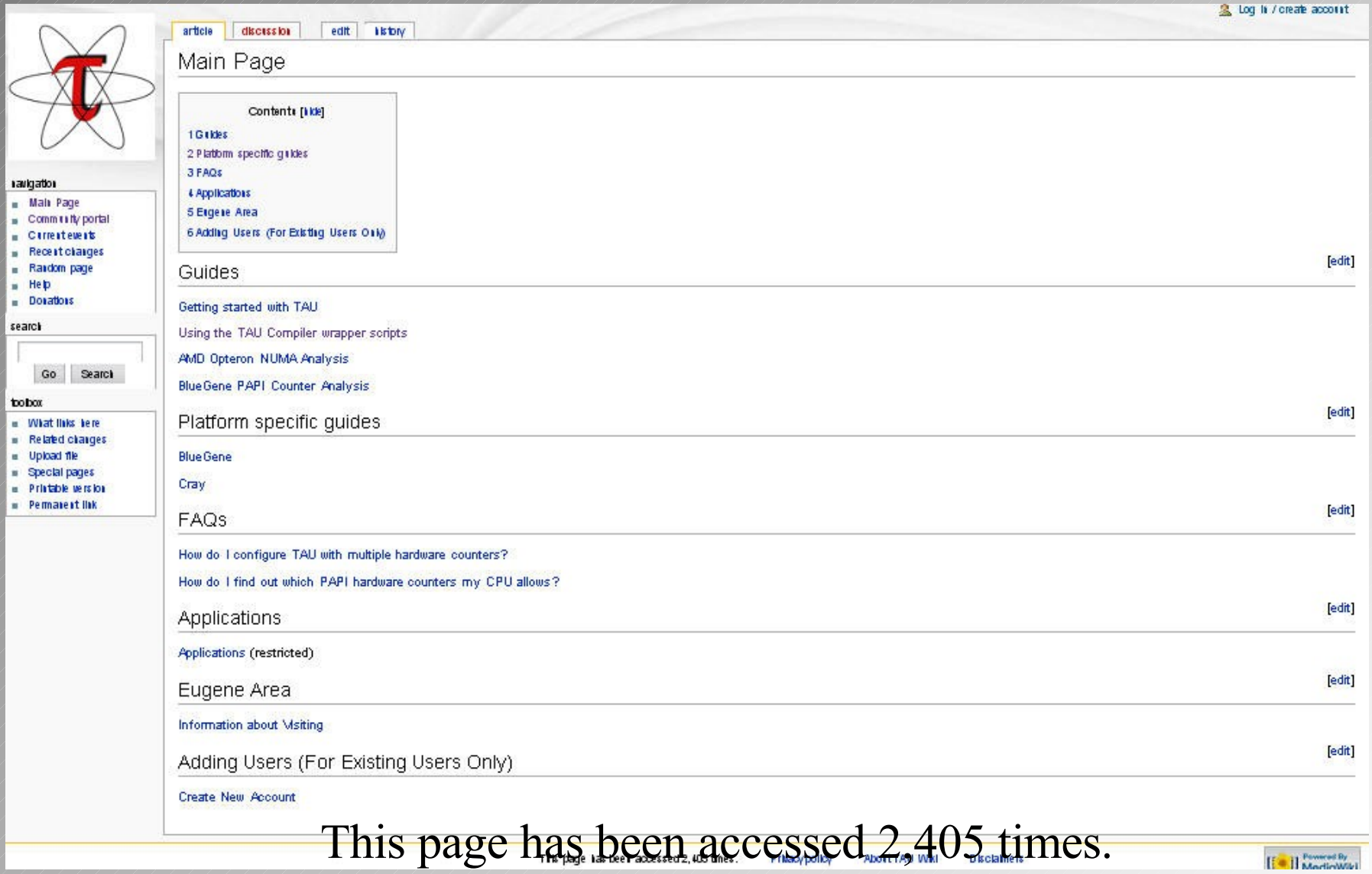
-> TraceTAUex TraceWrite include lib meins start.bat testdir

Feedback (5)

- Aber ich gebe nicht so schnell auf und beginne mich einzuarbeiten
- Dank Julian habe ich ja auch viele Hinweise bekommen, gerade bei der Einarbeitung in den Cluster (Shell-Box)
- Auch schon die Links zum lesen, aber leider:
  - Alles Englisch
  - Und alternative Quellen sind quasi nicht zu finden ...
- Aber zum Glück waren ja Ferien und ich hatte Zeit

Feedback (6)

- OTF? =
  - O n T h e F l i
    - das ultimative Config- & Statustool für fli4l
  - PostScript-flavoured OpenType
    - (Datei-Endung .otf)
  - Und noch viele mehr ...
    - Gesucht war ja gewesen Open Trace Format



This page has been accessed 2,405 times.

Während meines Praktikums wurde die Seite endlich seit vielen Monaten wieder bearbeitet! Am Anfang war diese Quasi leer gewesen ... Jetzt hat sie schon stolze 14? Seiten ^^

Veraltete und fehlerhafte Webseiten ...
Wohlgemerkt als einzige Bezugsquelle!

Feedback (9)

- Aber ich durfte und konnte ja Julian per ICQ immer wieder mit Fragen löchern und bekam gute Hilfe
- Alleine ohne die Hilfe hätte ich das vielleicht nicht geschafft
- Deswegen vielen dank noch einmal :)

Feedback - Abschlussfeststellungen

- Bin total Windowsverwöhnt
- In einer Shell-Box(DOS) kann ich jetzt auch wieder überleben
- Zum Glück gibt’s in der Shell-Box auch Copy & Paste
- Zum Glück gibt’s auch –dump, sonst hätte ich selbst nichts sehen können, was ich gemacht habe
- Nerviges regelmäßiges Disconnect des SFTP-Clienten trotz „keep alive“
- Heimarbeitsmöglichkeit war sehr angenehm!
- Im Zweifelsfall möglichst viele Fehler im Code einbauen ^^

Quellen

- http://www.cs.uoregon.edu/ [TAU]
- http://www.nic.uoregon.edu/ [TAU]
- http://www.paratools.com [TAU & OTF]
- http://www.google.de [Suchen]
- http://www.cplusplus.com/ [C++ Code]
- Julian Martin Kunkel

Ende

So das war es dann ^^

Vielen Dank fürs Zuhören!