TECHNISCHE UNIVERSITÄT DRESDEN

Center for Information Services and High Performance Computing (ZIH)

[Parallel] Debugging

14. Dezember 2009

INF 1038
Nöthnitzer Straße 46
01187 Dresden
0351 - 463 38474

Thomas William

ZIH
Center for Information Services & High Performance Computing

Foliensatz

Verfügbarkeit der Folien

Vorlesungswebseite:
`http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/lehre/ws0910/lctp`

Inhalt

1. Bugs - Wie man Wanzen findet
   - Serielle Fehler
   - HeisenBugs
   - Parallele Fehler
2. Debugger
3. Bugs verhindern
4. Das Heat-Beispiel - Einführung
5. DDT - ein Debugger für parallele Programme
   - Debuggen von seriellen Programmen
   - Debuggen von multithreaded Programmen
   - Debuggen von multiprocess/hybrid Programmen
6. Marmot - MPI Korrektheitsprüfung

Ausgangssituation

"Das Programm tut nicht was es soll"

- Anormaler Programmabbruch
- Falsche Ergebnisse
- Unverständliches/unvorhersagbares Verhalten

Beispiel:

```
gcc myprog.c -o myprog
./myprog
Segmentation fault
```

Was nun?

"Fehlerquelle finden"

Speichermanagement, algorithmischer Fehler, Bibliotheken . . .

- Sogenanntes printf-debugging
- *trace (strace, ptrace, mtrace, ltrace)
- Debugger
- . . .

Bugs

Wie man Wanzen findet

Fehlerarten

- Syntaxfehler
- Laufzeitfehler
  - "Normale" serielle Fehler
  - "Heisenbugs"
  - Parallele Fehler
- Logische Fehler
- Designfehler
- Regressionsbugs
- Fehler im Bedienkonzept
- . . .

Serielle Fehler

Syntax Fehler

```
1 int main (int argc, char **argv)
2 {
3     int i
4     return 0;
5 }
```


Syntax Fehler

```
1 int main (int argc, char **argv)
2 {
3     int i
4     return 0;
5 }
```

Zeile 3: Fehlendes ";"

Speicherzugriffsfehler

Ungültiger Zugriff

```
1 int *p;
2 p[100] = 123;
```

Undefinierter Lesezugriff

```
1 int i,j;
2 for (i = 0; i < j; i++)
```


Speicherzugriffsfehler

Ungültiger Zugriff

```
1 int *p;
2 p[100] = 123;
```

Zeile 2: Für den Zeiger p wurde kein Speicher allokiert.

Undefinierter Lesezugriff

```
1 int i,j;
2 for (i = 0; i < j; i++)
```

Zeile 2: Wert von j nicht definiert

Arithmetic Errors

```
1 int x, y=1000000;
2 x = y*y;
```


Arithmetic Errors

```
1 int x, y=1000000;
2 x = y*y;
```

Zeile 2: 32bit Integer Überlauf in `x`

Portability Errors

```
1 int x, y;
2 x = (int)(&y);
3 *((int*)x)=55;
```


Portability Errors

```
1 int x, y;
2 x = (int)(&y);
3 *((int*)x)=55;
```

Zeile 2: Verlustfrei nur auf 32bit Systemen

Unendliche Schleifen

```
1 while (1 == 1)
2       printf ("Hello");
```


Unendliche Schleifen

```
1 while (1 == 1)
2       printf ("Hello");
```

Zeile 1: keine Abbruchbedingung

Speicherlecks

```
1 int do_work (int size)
2 {
3       int *x;
4       x = (int*) malloc (sizeof(int) * size);
5 }
```


Speicherlecks

```
1 int do_work (int size)
2 {
3       int *x;
4       x = (int*) malloc (sizeof(int) * size);
5 }
```

Zeile 5: Speicher wird nicht freigegeben

Serielle Fehler

Fehlerhafte Nutzung von Bibliotheksaufrufen

```
1 FILE *file;
2 file = fopen ("myfile.txt", "rwa");
```


Serielle Fehler

Fehlerhafte Nutzung von Bibliotheksaufrufen

```
1 FILE *file;
2 file = fopen ("myfile.txt", "rwa");
```

Zeile 2: "rwa" nicht definiert

Serielle Fehler

Genereller semantischer Fehler

```
1 int i=0,k=0;
2 int a[100], b[100], c[100];
3 for (i = 0; i < 100; i++)
4     c[k] = a[k] * b[k];
```


Serielle Fehler

Genereller semantischer Fehler

```
1 int i=0,k=0;
2 int a[100], b[100], c[100];
3 for (i = 0; i < 100; i++)
4     c[k] = a[k] * b[k];
```

Zeile 4: Vom Algorithmus her sollte in der Schleife über `i` indiziert werden.

Begriffsklärung

Fehler der nur unter bestimmten (i.d.R. unbekannten) Randbedingungen auftritt.

- Schwer zu finden da nicht deterministisch / nicht reproduzierbar
- Im schlimmsten Fall verändert ein Debugger diese Randbedingung

Mögliche Ursachen:

- Nutzung uninitialisierter Speicherbereiche
- Race Conditions

Beispiel

```
1 int a[100], i;
2 int *p;
3 for (i = 0; i < 100; i++){
4     if (a[i] == 1234)
5         p[i] = 5678;
6 }
```


Beispiel

```
1 int a[100], i;
2 int *p;
3 for (i = 0; i < 100; i++){
4     if (a[i] == 1234)
5         p[i] = 5678;
6 }
```

Zeile 4: a[i] wird nicht initialisiert. Der Inhalt des Feldes ist damit zufällig. Die if-Abfrage (`a[i] == 1234`) könnte daher zufällig erfüllt sein.

Parallele Fehler

- Serielle Fehler treten auch in parallelen Programmen auf
- Threads/Prozesse erzeugen neue Fehlerquellen

Beispiele:

- Races
- Deadlocks
- Falsche Nutzung des Thread Paradigmas

Race

Bei einer Race Condition ist das Programmverhalten von der Abarbeitungsfolge der Threads/Prozesse abhängig.

OpenMP-Beispiel

```
1 int x,y;
2 #pragma omp parallel
3 {
4     x = omp_get_thread_num ();
5     #omp barrier
6     #omp master
7         printf ("Master is:%d" ,x);
8 }
```


Race

```
1 int x,y;
2 #pragma omp parallel
3 {
4     x = omp_get_thread_num ();
5     #omp barrier
6     #omp master
7         printf ("Master is:%d" ,x);
8 }
```

Zeile 4: Da alle Threads auf x schreiben entsteht eine "write-write race". Der letzte Thread der auf x schreibt gewinnt den Wettlauf, d.h. dessen Wert wird letztendlich in x gespeichert.

Race als Grund für einen Heisenbug

```
1 int x,y;
2 #pragma omp parallel
3 {
4     #omp master
5         sleep(5);
6     x = omp_get_thread_num ();
7     #omp barrier
8     #omp master
9         printf ("Master is:%d" ,x);
10 }
```


Race als Grund für einen Heisenbug

```
1 int x,y;
2 #pragma omp parallel
3 {
4     #omp master
5         sleep(5);
6     x = omp_get_thread_num ();
7     #omp barrier
8     #omp master
9         printf ("Master is:%d" ,x);
10 }
```

Durch das `sleep()` wird der Wert von x **meistens** durch den *master* selbst bestimmt.

Deadlock

Ein Deadlock entsteht durch eine zyklische Abhängigkeit im System bei dem zwei oder mehr parallele Einheiten zum (unendlichen) Warten gezwungen werden.

Deadlock: OpenMP-Beispiel

```
1 #pragma omp parallel sections
2 {
3     #omp section
4     {
5         omp_set_lock(&lock_a);
6         omp_set_lock(&lock_b);
7         omp_unset_lock (&lock_b);
8         omp_unset_lock (&lock_a);
9     }
10    #omp section
11    {
12        omp_set_lock(&lock_b);
13        omp_set_lock(&lock_a);
14        omp_unset_lock (&lock_a);
15        omp_unset_lock (&lock_b);
16    }
17 }
```


Deadlock: OpenMP-Beispiel

```
1 #pragma omp parallel sections
2 {
3     #omp section
4     {
5         omp_set_lock(&lock_a);
6         omp_set_lock(&lock_b);
7         omp_unset_lock (&lock_b);
8         omp_unset_lock (&lock_a);
9     }
10    #omp section
11    {
12        omp_set_lock(&lock_b);
13        omp_set_lock(&lock_a);
14        omp_unset_lock (&lock_a);
15        omp_unset_lock (&lock_b);
16    }
17 }
```

- Thread1 hat `lock_a` und wartet auf `lock_b`
- Thread2 hat `lock_b` und wartet auf `lock_a`
- Zyklische Abhängigkeit $\Longrightarrow$ Deadlock

Debugger

Debugger - Grundlegende Eigenschaften

- Analyse eines laufenden Programms
  - Variablen ausgeben
    (skalare Typen, Felder, Strukturen, abgeleitete Typen, Klassen)
  - Aktuelle Codezeile der Abarbeitung, Function Call Stack
- Eingriff in den Programmablauf
  - Anhalten des Programms an beliebiger Zeile im Code (Breakpoint)
  - Anhalten des Programms bei bestimmtem Variableninhalt
    - Conditional Breakpoint
    - Watchpoint
  - Anhalten des Programms bei Fehler/Abbruch
  - Schrittweise Ausführung Zeile-für-Zeile (Stepping)

Function Call Stack

Alle aktiven Funktionen eines Prozesses welche gerufen aber noch nicht beendet wurden.

Weitere Eigenschaften bestimmter Debugger

- Unterstützung paralleler Anwendungen
- Graphische Analyse der Datenstrukturen
- Visualisierung von 3D-Feldern
- Speicher-Debugging
- Veränderung des Variableninhalts zur Laufzeit
- Rückwärtsausführung der Anwendung

Debugger - eine Auswahl

GNU Debugger	gdb	cmdline	Threads
GNU Data Display Debugger	ddd	graphical	Threads
Allinea Distributed Debugging Tool	ddt	graphical	Threads+MPI
Totalview	totalview	graphical	Threads+MPI
Valgrind	valgrind	cmdline	Threads
DUMA	duma	library	possible

DUMA Begriffsklärung

DUMA (Detect Unintended Memory Access) ist Fork des Electric-Fence (Bibliotheken zum Speicherdebuggen) und wird von den Pixar Studios seit den 80er Jahren für ihre Rendersoftware eingesetzt

Debugger - prinzipielle Nutzung

1. Quellcode mit dem "-g" Flag neu übersetzen

```
gcc -g code.c -o prog.exe
```

2. Programm unter Kontrolle des Debuggers starten

```
ddt ./prog.exe
```

   - Programm bis zum Absturz laufen lassen
   - Function Call Stack zum Zeitpunkt des Absturzes untersuchen

3. Quellcode anpassen
4. Gehe zu 1.

Operationsmodi des Debuggers

1. Programm unter Kontrolle des Debuggers starten
   - Standardweg
2. Debugger an laufenden Prozess anhängen
   - Lang laufende Anwendungen etc.
3. Core-Dateien im Debugger laden
   - Core schreibt Speicherabbild und Registerzustand wenn das Programm abstürzt
   - Muss mit

     ```
     ulimit -c corefilesize
     ```

     vorher aktiviert werden
   - Nützlich bei langen Anwendungen die "eigentlich" stabil laufen
   - Allerdings liefert die post-mortem Analyse nur statische Informationen

Was gibt es sonst noch an Debug-Möglichkeiten

Memory-Debugger

- Erkennen von Fehlern bei dynamischem Speichermanagement
  $\Longrightarrow$ Valgrind, DUMA

Correctness-Checker

Threads

- Falsche Thread Parallelisierung (Deadlock etc.)
  $\Longrightarrow$ Intel Thread Checker

Prozesse

- Falsche MPI Nutzung
  $\Longrightarrow$ MARMOT

Memory-Debugger

Speicherzugriffsfehler treten manchmal stark verzögert auf
(nach dem fehlerhaften Code)

- Memory Debugger helfen dann den wahren Grund für den Fehler zu finden
- Hauptsächlich werden Fehler im Speichermanagement gesucht:
  - Zugriff auf nicht allokierte Bereiche
  - Zugriff über Grenzen der Speicherstruktur hinweg
  - Speicherlecks
  - ...

Grundsätzlich gibt es aktuell zwei bekannt Ansätze:

- **Valgrind**: Simulation des Programms in virtueller Maschine
- **Bibliotheken** (Electric Fence, Dmalloc, DUMA): Speichermanagement durch Bibliothek ersetzen

Was kann man tun um Bugs zu verhindern?

- Erst denken, dann programmieren, nicht "hacken"
- Auf das Schreiben des Codes konzentrieren, nicht das Ver- / Entschlüsseln
  - Kommentare (wie und warum, nicht was programmiert wird)
  - Code-Style-Guides befolgen, sinnvolle Namen etc
  - Einheitliche Formatierung (code beautifier)
  - . . .
- Programmiertechniken zur Codeverbesserung nutzen
  - Code Review
  - Pair Programming
  - Pre/Post Condition Checks (Beispiel: `assert(pointer != NULL)`)
- Von Beginn an einen "Verbose" Modus vorsehen (Debug-Ausgabe)
  - Permanentes, im Code abschaltbares printf-Debugging vorsehen

Beispiel debug-printf

dprintf - Teil1

```
#define DEBUG_NOTHING -1
#define DEBUG_ERROR 0
#define DEBUG_WARNING 1
#define DEBUG_INFO 2

#ifndef DEBUG
#  define DEBUG 0
#endif
```


Beispiel debug-printf

dprintf - Teil2

```
int dprintf( const int level, const char * comment, ... ){
  int result = 0;

#if defined( DEBUG )
  va_list args;
  va_start( args, comment);

  if ( DEBUG >= level){
 result = vfprintf( stderr, comment, args );
}

#else
  /* avoid compiler-warnings about unused parameters
  void * dummy = 0;
  dummy = (void *) ( &level );
  dummy = (void *) ( comment ); */

#endif
  fflush(stderr);
  fflush(stdout);
  return result;}
```


Was wenn doch Bugs auftreten?

Maßnahmen ergreifen um Bug frühstmöglich zu erkennen

- Compiler Flags: `-Wall, -pedantic(-errors)` . . .
- Werkzeuge nutzen
  - Fremden Code Schritt-für-Schritt im Debugger analysieren
  - Memory Checker (Valgrind) nutzen
- Für parallele Programme
  - **OpenMP**: Intel Thread Checker
  - **MPI**: Marmot

Intel Compiler

Allgemeine Flags:

- **-g** (Debug Informationen)
- **-O0** (keine Code-Optimierung)

Intel C Compiler - Compile-Zeit Information:

- **-Wall** ((fast) alle Warnungen)
- **-Wp64** (Diagnoseinformationen für 64-bit Portierung)
- **-Wuninitialized** (uninitialisierte Variablen checken - unsicher!)
- **-strict-ansi** (strikt ANSI C/C++)

Intel Fortran Compiler - Compile-Zeit Informationen:

- **-warn all** (alle Warnungen)
- **-std90 / -std95** (striktes Fortran 90/95)

Laufzeit Informationen:

- **-traceback** (Call Stack Traceback im Fall schwerwiegender Fehler)
- **-C** (Laufzeit Checks z.B.: Feldgrenzen, uninitialisierte Variablen)
- **-fpe0** (Abbruch bei Fließkomma Exceptions)

Das Heat-Beispiel

Beispiel Code für die Übungen

Das Heat-Beispiel - Einführung

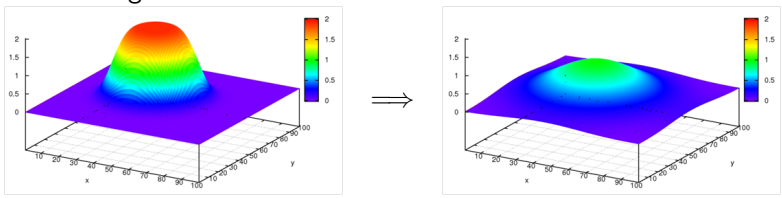
- Löser für die Wärmeleitungsgleichung
- Gleichung beschreibt die Hitzeverteilung über der Zeit
- Gleichung (2D):

$$\frac{\partial u}{\partial t} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

- Beispiel implemeniert die Lösung auf einem 2D-Gitter
- Einzelner Zeitschritt für eine Zelle im Gitter:

$$\frac{\Delta u[x,y]}{\Delta t} = k\left(\frac{u[x-1,y]+u[x+1,y]-2u[x,y]}{\Delta x^2} + \frac{u[x,y-1]+u[x,y+1]-2u[x,y]}{\Delta y^2}\right)$$

- Visualisierung als 3D Gitter:

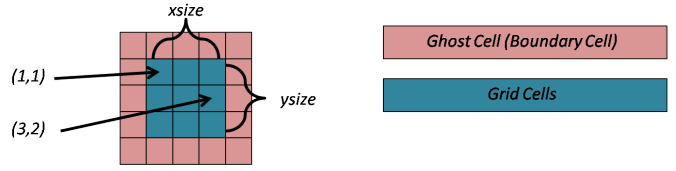


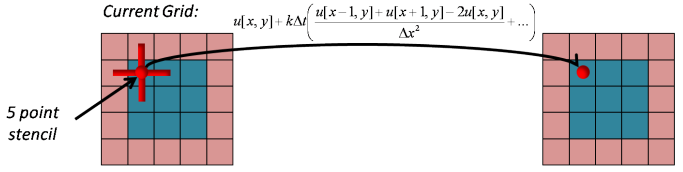
Das Heat-Beispiel - Quelltext

- Verfügbar in C und Fortran90
- 3 Versionen: Seriell, OpenMP und MPI
- Wichtigstes Element ist die Datenstruktur für das 2D-Gitter
- Funktionen:
  - **heatAllocate & heatDeallocate** - De/Allokation des Gitters
  - **heatInitialize & heatInitFunc** - Hitzerverteilung zu Beginn
  - **heatPrint & heatOutput** - Ausgabe ind Datei oder stdout
  - **heatTimestep** - Ein Zeitschritt (ganzes Gitter)
  - **heatBoundary** - Grenzdaten übertragen
  - **heatTotalEnergy** - Gesamtenergie des Systems
  - **Main function** - Hauptschleife

Das Heat-Beispiel - Gitter

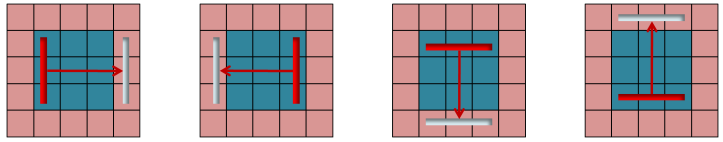
- Die Gitterstruktur enthält 2 Gitter und die Größenangabe
- Die Geisterzellen werden als Nachbarn für Grenzzellen benötigt
- Es werden jeweils für alle Zellen der aktuelle Schritt berechnet und dann im zweiten Gitter gespeichert
- Danach werden die beiden Gitter getauscht
- Gitter:





Das Heat-Beispiel - Geisterzellen (ghost cells)

- Jede Zelle braucht eine 5-Zellen Schablone (Stencil): links, rechts, oben, unten
- Die Geisterzellen dienen als Nachbarn
- Werte können konstant sein oder aus dem Netz errechnet werden
- Bei jedem Zeitschritt müssen über eine Kopieroperation die Geisterzellen aktualisiert werden
- In der `heatBoundary`-Funktion implementiert



Das Heat-Beispiel - Ablauf

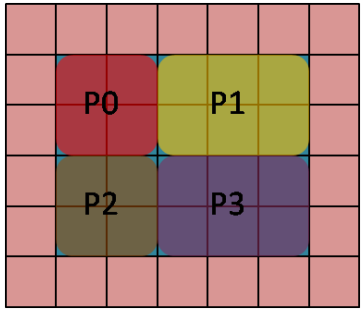
- Initialisierung
- Iteration über Zeitschritte und Grenzdatenaustausch (`heatTimestep`, `heatBoundary`)
- Jeder Zeitschritt liefert die maximale Temperaturdifferenz zurück
- Die Iteration stoppt wenn die Differenz unter einen Grenzwert fällt
- Sowohl die Start- als auch die Endkonfiguration der Gitter werden ausgegeben
- 4 Zwischenstände der Hitzeverteilung werden jeweils in eine Datei geschrieben: "`data-<#Iterations>.dat`"
- Ein beigelegtes Gnuplot Skript visualisiert diese Daten

Das Heat-Beispiel - OpenMP Version

- Sehr einfache Umsetzung
- Beide Gitter werden komplett im "shared memory" gehalten
- Die Berechnung der Zeitschritte geschieht in Threads
- Es gibt keine "data races" da die Ergebnisse jeweils in das zweite Gitter geschrieben werden
- Die Berechnung der maximalen Temperaturdifferenz muss angepasst werden

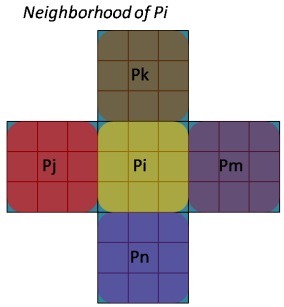
Das Heat-Beispiel - MPI-Version

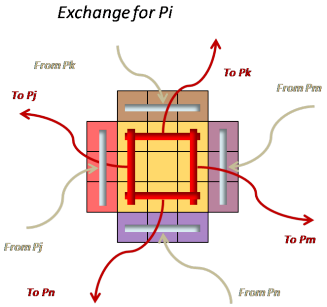
- Jeder Prozess hat seine eigene Kopie der beiden Gitter
- Initialisierung etc. erfolgt auf allen Prozessen separat
- Jeder Prozess berechnet einen Teil des Gitters
- Mit Hilfe der MPI-Funktionen wird ein kartesisches 2D-Gitter erstellt
- Kommunikation einer Daten-Reihe (Fortran) bzw. -Zeile (C) benutzt einen abgeleiteten Datentyp
  - Dabei müssen die neuen Grenzen ausgetauscht werden
  - Die Daten für die Ausgabe sind über mehrere Prozesse verteilt



Das Heat-Beispiel - MPI Version - Grenzdatenaustausch

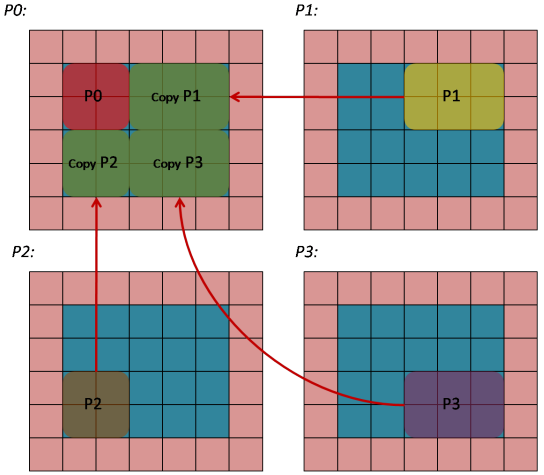
- Prozesse müssen Grenzdaten mit jeweiligen Nachbarn austauschen
- Ein kartesisches Gitter hat oben, unten, links und rechts Nachbarn
- Die Grenzen sind periodisch da ein periodisches kartesisches Gitternetz benutzt wird
  - Das 2D-Gitter ist eigentlich ein "Donut"
- Grenzdatenaustausch für Prozess "P_i"





Das Heat-Beispiel - Daten "einsammeln"

- Nach dem ersten Zeitschritt sind Daten über Prozesse verteilt
- Ausgabe entweder durch I/O von jedem Prozess oder durch sammeln der Daten auf einem Prozess
- Die Implementierung sammelt die Daten zur Ausgabe auf Prozess 0



Das Heat-Beispiel - Kompilieren

- Am Besten den Intel Compiler nutzen
- C:

```
cd debuggingC/serial
```

- Fortran:

```
cd debuggingF/serial
```

- Der Ordner enthält alle Beispielaufgaben zum seriellen Debuggen
- "exampleC.c / exampleF.F90" die fehlerfreie Version
- "exampleC-XX.c / exampleF-XX.F90" haben verschiedene Fehler
- Bauen der fehlerfreien Variante
  - C:

```
icc exampleC.c -o example.exe
```

  - Fortran:

```
ifort exampleF.F90 -o example.exe
```


Das Heat-Beispiel - Ausführen und Plotten

- Ausführen:

```
bsub -n 4 -Is -W 2:00 bash #mars
./example.exe #lokal/mars
```

- Es werden initiale und finale Gitterzustände sowie Energielevels ausgegeben
- Die Dateien `data-XXXXXXXX.dat` werden erstellt
- Jetzt können die Daten visualisiert werden:

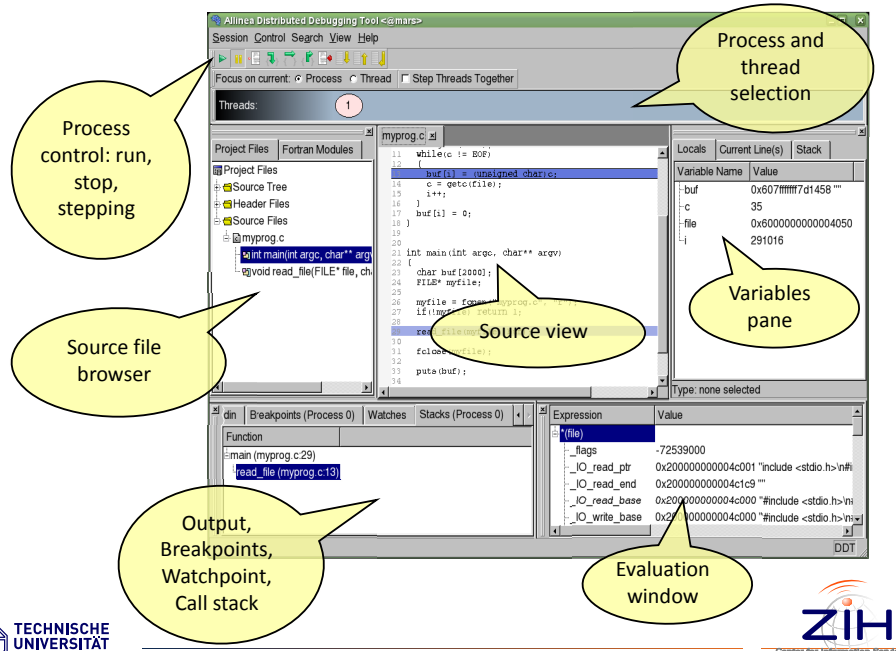
```
bash build-gnuplot.sh
gnuplot gnuplot.ini
gv heat-conduction.ps
```

- Gnuplot sollte eine Animation von 4 Zeitschritten anzeigen die dann auch als ps-Datei genutzt werden kann

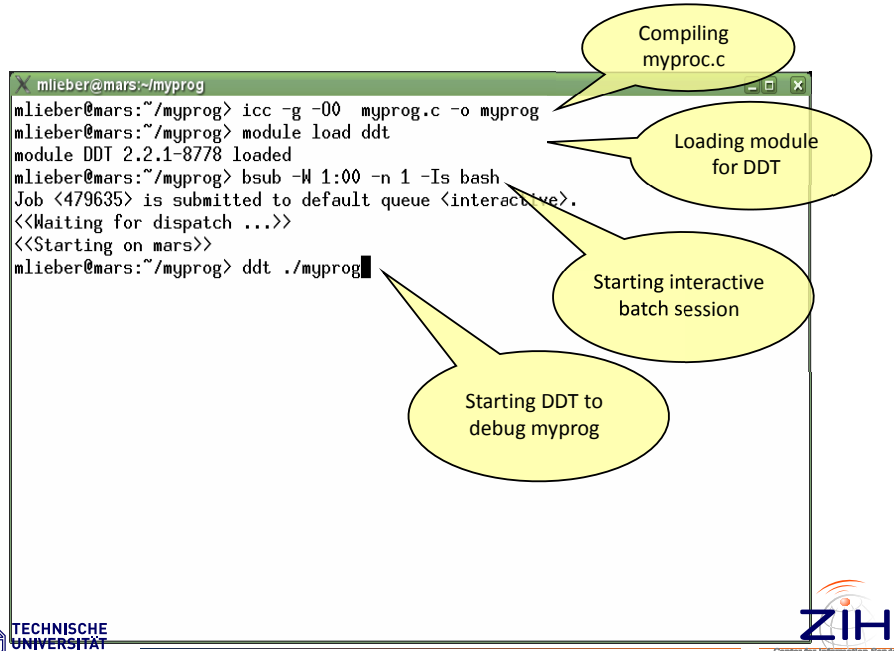
Allinea Distributed Debugging Tool (DDT)

Ein Debugger für parallele Programme

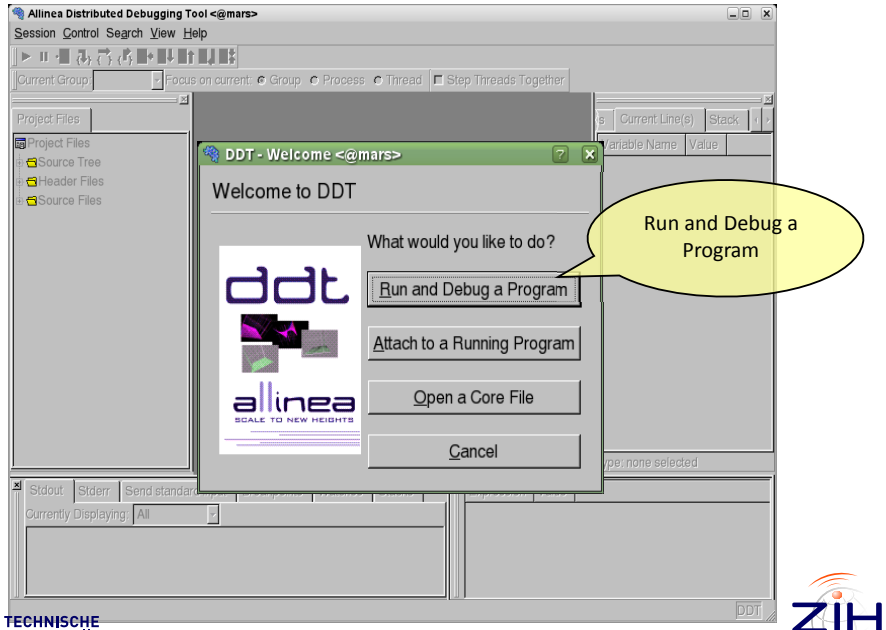
Allinea Distributed Debugging Tool (DDT)



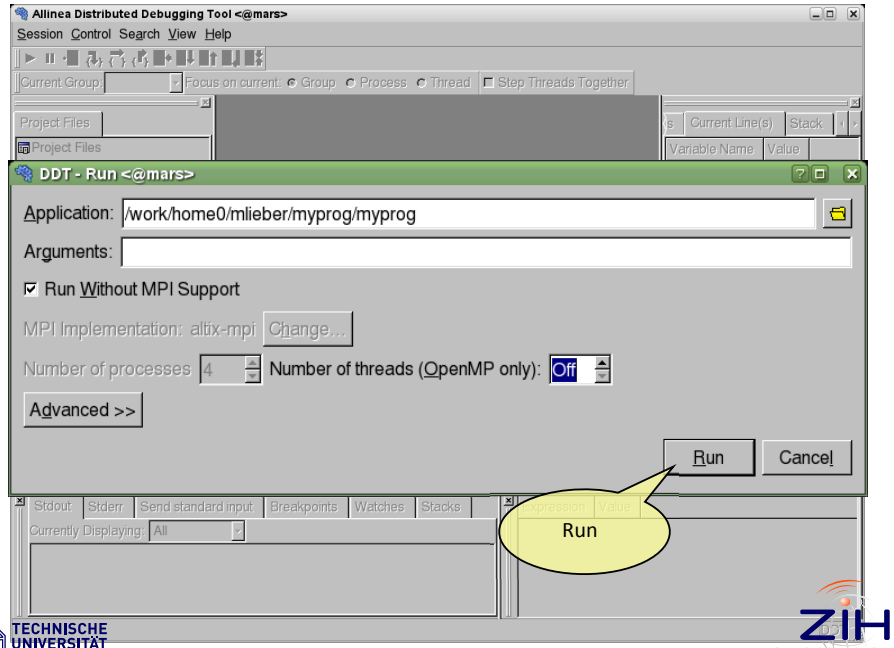
Example: Debugging mit dem DDT auf `mars`



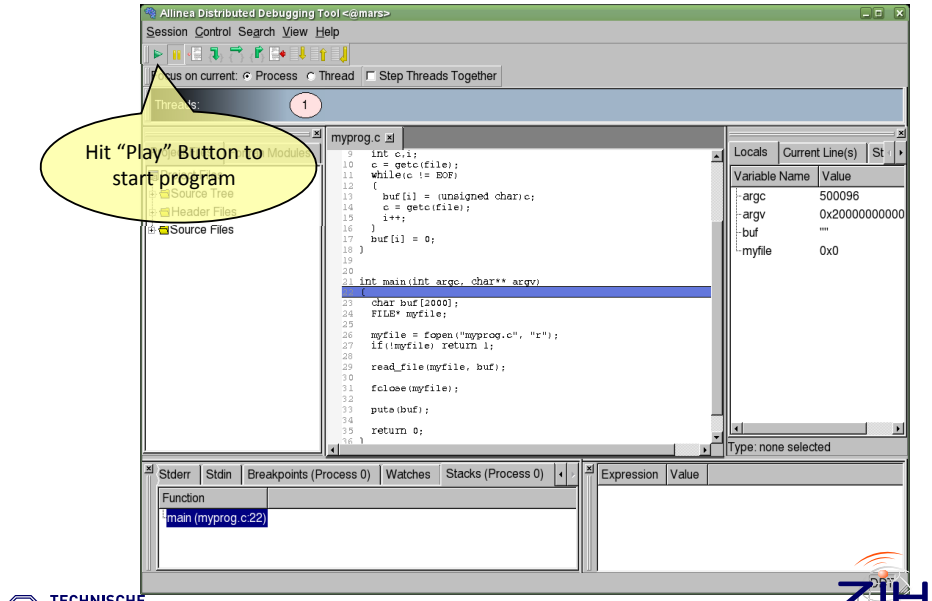
Example: Debugging mit dem DDT auf `mars`



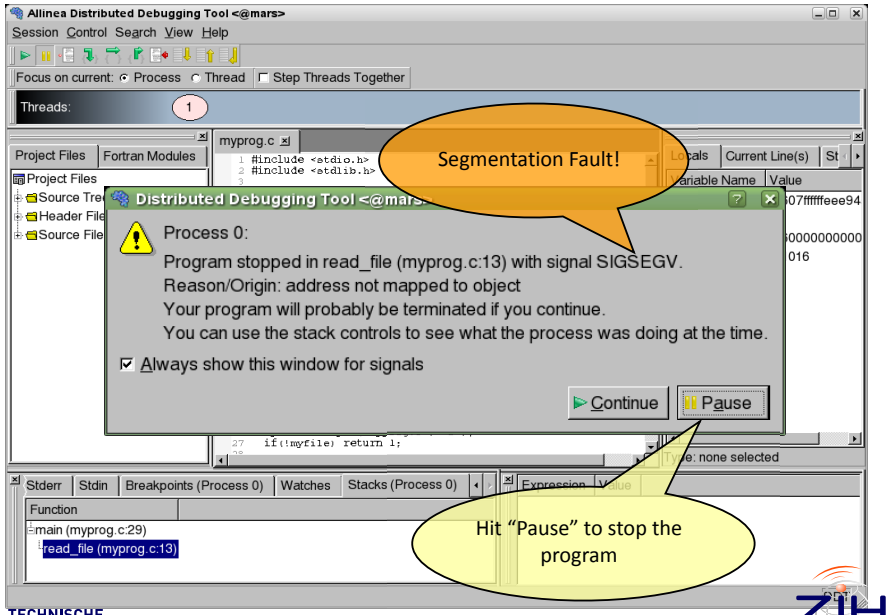
Example: Debugging mit dem DDT auf mars



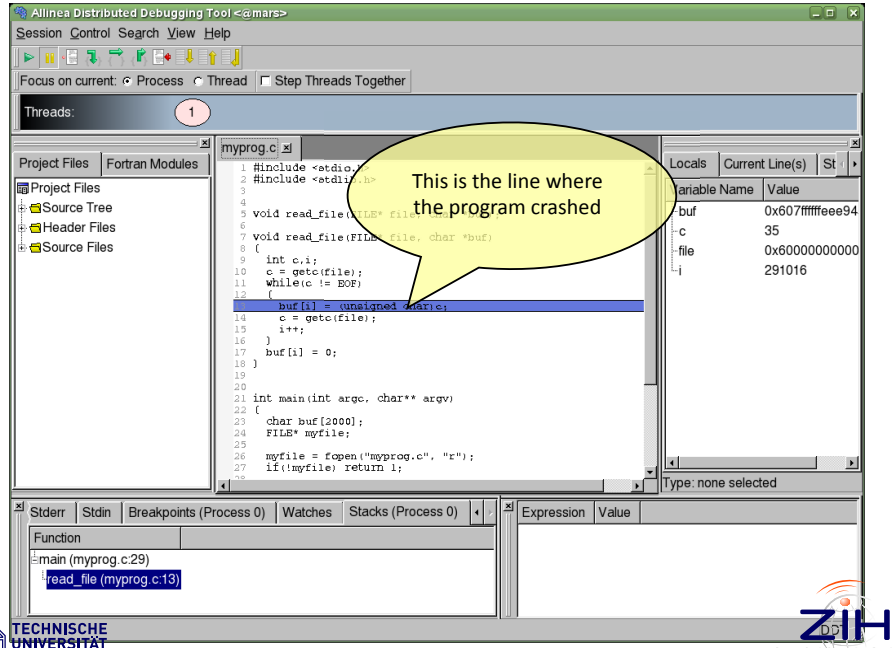
Example: Debugging mit dem DDT auf mars



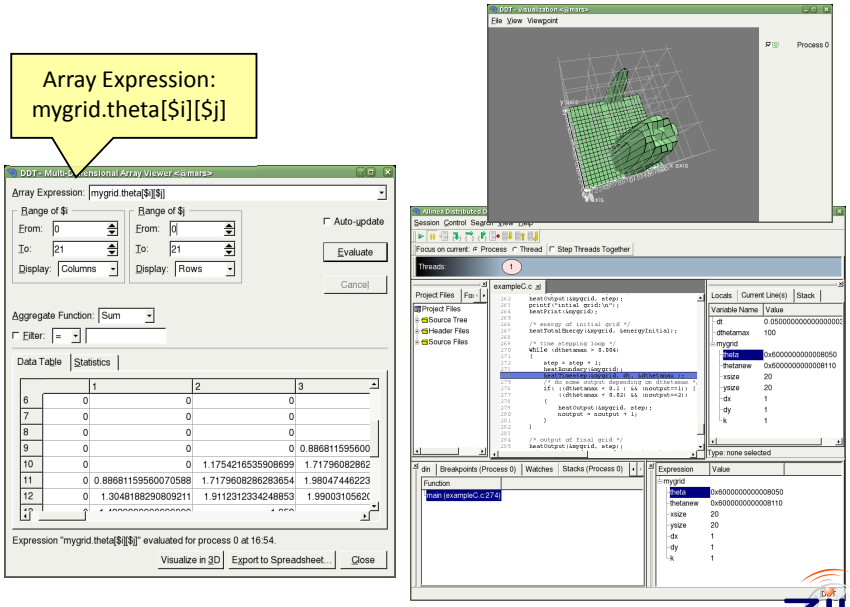
Example: Debugging mit dem DDT auf mars



Example: Debugging mit dem DDT auf mars



Datenfelder in C



Conditional Breakpoints & Memory Debugging im DDT

Conditional Breakpoints

- Breakpoint hinzufügen (Rechtsklick oder Doppelklick auf die Zeile)
- Den Reiter für "breakpoints" öffnen
- Die Bedingung in der "condition" Spalte Eintragen (z.Bsp. i==10)

Memory Debugging

- Im Startbildschirm unter "advanced" Enable Memory Debugging einschalten
- In den "settings" das "medium" Level und "add guard pages" auswählen

DDT-Live-Demo

Beispiel: firstC

- Kompilieren (`icc -g -O0 firstC.c -o first.exe`)
- Starten des Programms, danach mit ddt
- Stepping
- Programm Absturz
- Core dump erzeugen (`ulimit -c 100000`)

Beispiel: exampleC

- Breakpoints
- Call stack
- Conditional breakpoints (Iteration 100)
- Variables (`dthetamax`)
- Felder (`theta`)
- Felder visualisieren (`mygrid.theta[$i][$j]`)

Beispiel: exampleC-05

- An laufenden Prozess anhängen

DDT kennen lernen

- "firstC" oder "firstF"
  - Mit dem ddt vertraut machen
  - Oberfläche kennen lernen
- "exampleC" oder "exampleF"
  - Nutze einen "conditional breakpoint" um den Wert von `dthetamax` nach 100 Schritten zu ermitteln
  - Finde heraus in welchem Schritt der Wert der Zelle theta(15,10) größer als 0 wird

Debuggen von multithreaded Programmen

Debuggen von multithreaded Programmen

Typische multi-threaded Bugs:

- Data Races
- Deadlocks
- Falsche Nutzung der Paralleliserungsstrategie

Programm kann laufen und trotzdem Fehler in Form von Races enthalten

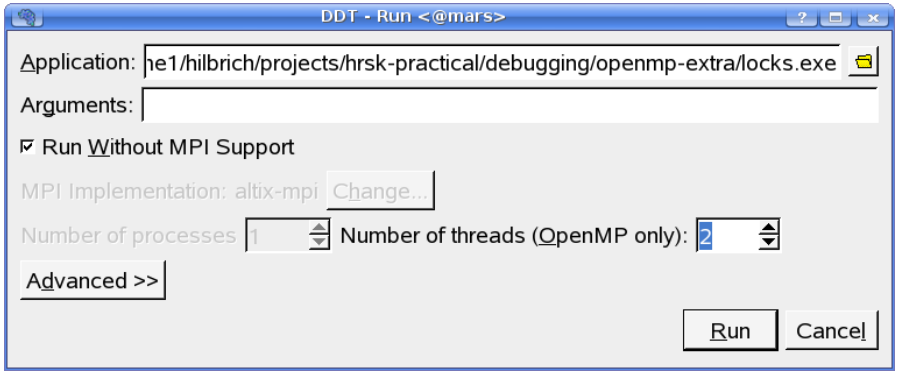
- Heisenbugs treten nicht immer auf . . .

Techniken:

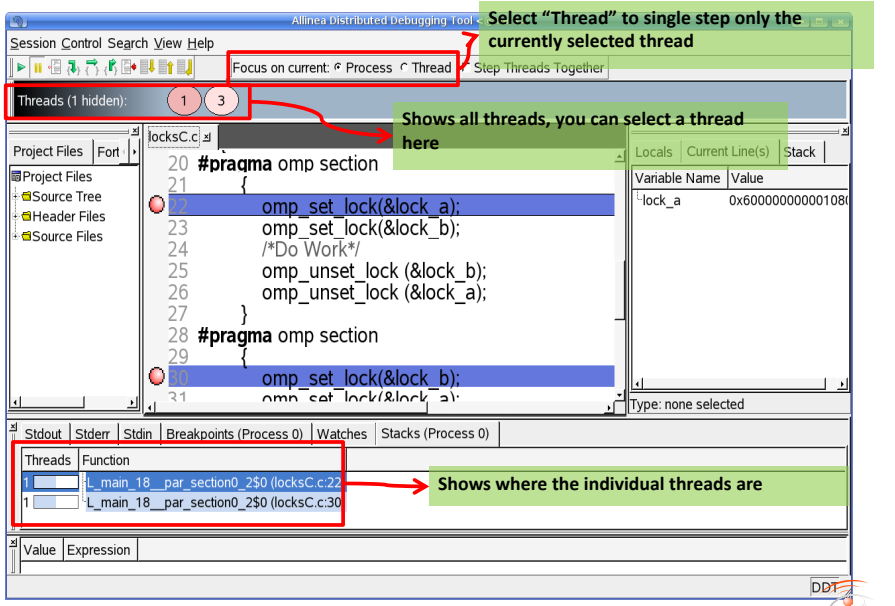
- Anzahl der shared Variablen minimieren: `#pragma omp parallel default(none)`
- Nutzung der shared Variablen doppelt prüfen
- Tools zur Quelltext-Validierung nutzen

DDT und Threads

- DDT starten
- Im "Run" Fenster Anzahl der Threads einstellen



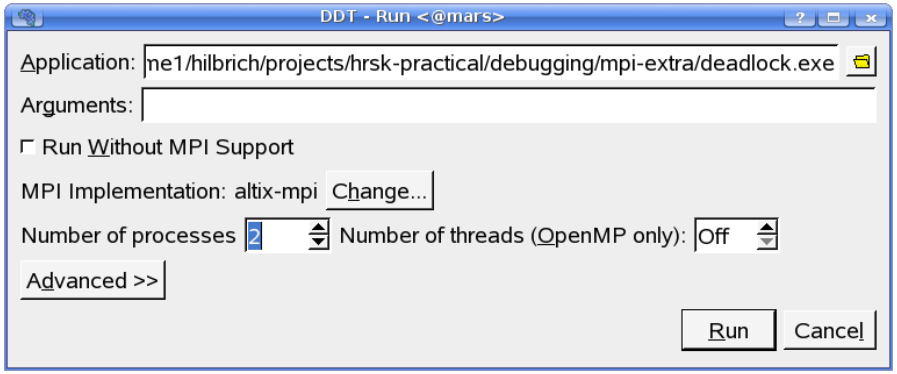
DDT und Threads



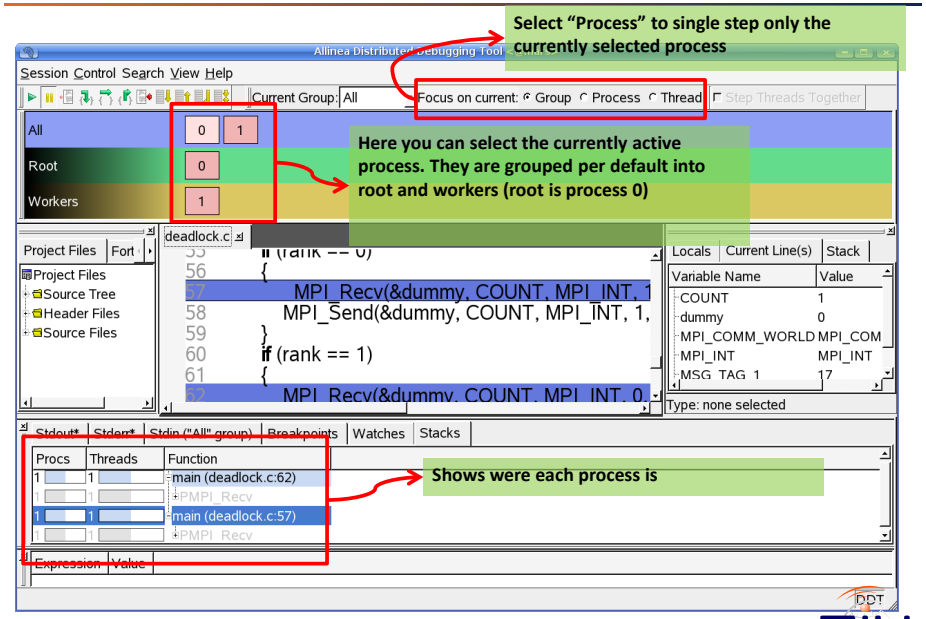
Debuggen von multiprocess/hybrid Programmen

DDT und MPI

- DDT starten
- Im "Run" Fenster Anzahl der Prozesse einstellen



DDT und MPI



MARMOT Prüfen von MPI-Programmen

Einführung

- Marmot dient zum prüfen der MPI-Aufrufe zur Laufzeit
  - Parameter
  - Standardkonformität
  - Überwachen der Resourcennutzung
- Ist eine in C++ geschriebene Bibliothek
- Wird einfach zum Programm hinzugelinkt
- Braucht einen eigenen Prozess und Debug-Symbole
- Unterstützt Fortran und C-Bindings des MPI 1.2 Standards
- Ergebnisse werden in ein Log geschrieben

Nutzung

Kompilation und Linken per Kompilerwrapper

- Wrapper ersetzen den eigentlichen Kompileraufruf
- C/C++: marmotcc oder marmotcxx
- Fortran: marmotf77 oder marmotf90
- Instrumentierung des Quelltextes erfolgt (meist) automatisch

Ausführen des Programms

- Es wird ein zusätzlicher Prozess gebraucht
- Anstatt `mpirun -np n` wird `mpirun -np n+1`
- Marmot erzeugt zusätzliche Last die das Verhalten beeinflussen können

Ausgabe

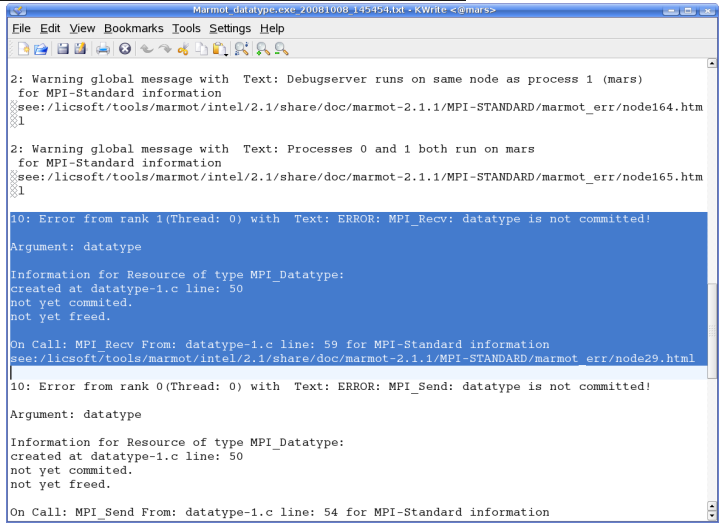
- Aktuell werden 3 Log-Formate unterstützt
  - ASCII
  - HTML
  - Cube (Scalasca(Kojak) GUI)
- Log-Type wird per Umgebungsvariable festgelegt

```
export MARMOT_LOGFILE_TYPE=...
```

- ASCII und HTML Logs sind meist unübersichtlich
- Am komfortabelsten ist die grafische Auswertung mit Cube

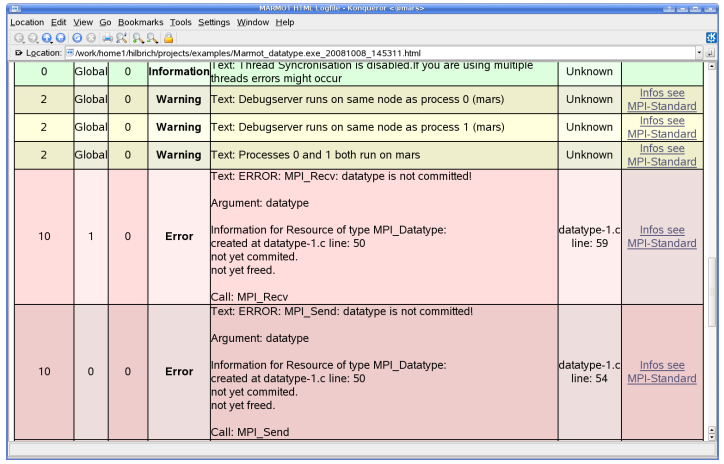
Beispiel Textausgabe

```
export MARMOT_LOGFILE_TYPE=0
```



Beispiel HTML-Ausgabe

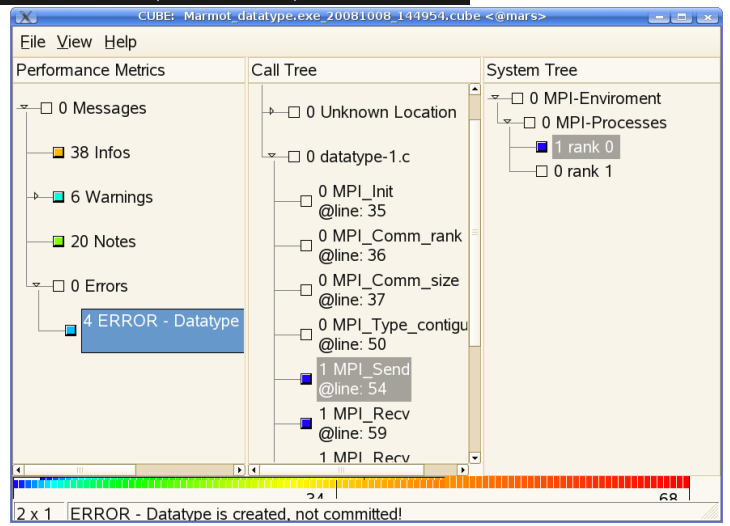
```
export MARMOT_LOGFILE_TYPE=1
```



0	Global	0	Information	Text: Thread Syncronisation is disabled.If you are using multiple threads errors might occur	Unknown	
2	Global	0	Warning	Text: Debugserver runs on same node as process 0 (mars)	Unknown	Infos see MPI-Standard
2	Global	0	Warning	Text: Debugserver runs on same node as process 1 (mars)	Unknown	Infos see MPI-Standard
2	Global	0	Warning	Text: Processes 0 and 1 both run on mars	Unknown	Infos see MPI-Standard
10	1	0	Error	Text: ERROR: MPI_Recv: datatype is not committed! Argument: datatype Information for Resource of type MPI_Datatype: created at datatype-1.c line: 50 not yet commited. not yet freed. Call: MPI_Recv	datatype-1.c line: 59	Infos see MPI-Standard
10	0	0	Error	Text: ERROR: MPI_Send: datatype is not committed! Argument: datatype Information for Resource of type MPI_Datatype: created at datatype-1.c line: 50 not yet commited. not yet freed. Call: MPI_Send	datatype-1.c line: 54	Infos see MPI-Standard

Beispiel Cube-Ausgabe

```
export MARMOT\_LOGFILE\_TYPE=2
```



Marmot - Live Demo - datatype.c

```
1  MPI_Init(&argc, &argv);
2  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3  MPI_Comm_size(MPI_COMM_WORLD, &size);
4
5  if (size < 2)fprintf(stderr, "This program needs at least 2 PEs!\n");
6  else{
7     sendBuf[0] = 1; sendBuf[1] = 2; sendBuf[2] = 3; sendBuf[3] = 4;
8     MPI_Type_contiguous(2, MPI_INT, &cont2Int);
9     MPI_Type_contiguous(2, cont2Int, &cont4Int);
10
11    if (rank == 0){
12         MPI_Send(sendBuf, 1, cont4Int, 1, MSG_TAG_1 ,MPI_COMM_WORLD);
13         MPI_Recv(recvBuf, 1, cont4Int, 1, MSG_TAG_2 ,MPI_COMM_WORLD,&
               status);
14    }
15    if (rank == 1){
16         MPI_Recv(recvBuf, 1, cont4Int, 0, MSG_TAG_1 ,MPI_COMM_WORLD,&
               status);
17         MPI_Send(sendBuf, 1, cont4Int, 0, MSG_TAG_2 ,MPI_COMM_WORLD);
18    }
19
20    MPI_Type_free (&cont4Int);
21    MPI_Type_free (&cont2Int);
22 }
23 MPI_Finalize();
```


Live Demo: Kompilieren, Linken and Ausführen

Kompilieren und Linken

- z.Bsp. auf Mars, Deimos

```
module load marmot
```

- Kompilieren

```
marmotcc datatype.c -o datatype.
    exe
```

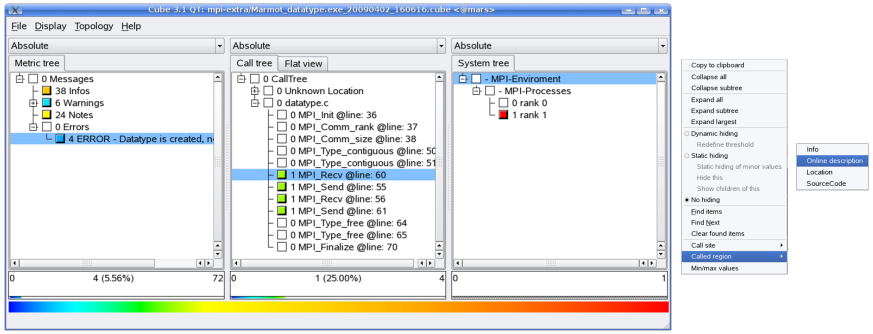
Marmot konfigurieren (optional)

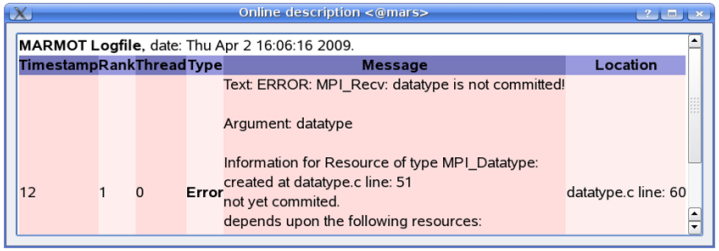
```
export MARMOT_LOGFILE_TYPE=2      # Cube Ausgabe
export MARMOT_LOG_FILTER_COUNT=2 # Fehler nur 2mal
```

Ausführen

```
mpirun -np 3 datatype.exe # 2 Anwendungsprozesse
cube3 Marmot_*.cube
```

Live-Demo: Cube GUI zur Auswertung





Wenn das automatische Instrumentieren fehl schlägt

Workaround:

```
#include <mpi.h>
#ifdef MARMOT
#include <enhancempicalls.h>
#endif
```

Beim Kompilieren muss dann `marmotcc -DMARMOT` verwendet werden

Marmot und Fortran

"marmotf77" oder "marmotf90"

- Der Quelltext wird instrumentiert (MPI-Aufrufe)
- Instrumentierung abschalten:

```
marmotfXX -MARMOT-noinst
```

- Marmot identifiziert die Fortran Version anhand der Dateiendung
- "fixed-source" Fortran:

```
marmotfXX -MARMOT-fixed
```

- "free source" Fortran:

```
marmotfXX -MARMOT-free
```


Marmot - wichtige Umgebungsvariablen

MARMOT_LOGFILE_TYPE:

- Ausgabeformat - 0=ASCII, 1=HTML, 2=Cube

MARMOT_LOG_FILTER_COUNT:

- Begrenzt die Anzahl der Ausgaben pro Fehlerart

MARMOT_LOGFILE_NAME:

- Name der Log-Datei

MARMOT_LOGFILE_PATH:

- Pfad der Log-Datei

MARMOT_LOG_FLUSH_TYPE:

- Zwingt MARMOT jeden Fehler sofort in Log-File zu schreiben
- Sinnvoll bei Programmabbrüchen

Literaturverweise + weiterführende Links

- GDB *http://www.gnu.org/software/gdb/*
- GDB CheatSheet *http://www.digilife.be/quickreferences/QRC/GDB Quick Reference.pdf*
- Link zu MARMOT: *http://www.hlrs.de/organization/amt/projects/marmot/*
- Wärmeleitungsgleichung: *http://en.wikipedia.org/wiki/Heat_equation*
- Sun Thread Analyzer: *http://developers.sun.com/sunstudio/downloads/ssx/tha*
- Helgrind: *http://valgrind.org/docs/manual/hg-manual.html*