Information and Communication Technologies (ICT) Programme

Project N^{o}: FP7-ICT- 247615

HEAP

Deliverable D3.6

Parallel code generator

User Manual and Tutorial

Author(s): Mihai T. Lazarescu, Luciano Lavagno (PoliTo)

Status -Version: V1.2

Date: 30 March 2012

Distribution - Confidentiality: Public

Code: HEAP_D3.6_PTO_V1.2_20120402.doc

Abstract:

This document describes the full HEAP flow for software parallelization, with the user-driven data dependency profile-driven approach. We provide a summary of the data dependency tracing and optimization, and then provide a detailed description of how this information should be used to derive a parallel software implementation. The operations of a new tool, which is included in the HEAP GUI to help the developer perform this last step, is also provided.

Disclaimer

The HEAP Consortium consists of the following companies:

No	Participant name	Participant short name	Country	Country
1	ST Microelectronics	STM	Co-ordinator	Italy
2	Synelixis Solutions Ltd	Synelixis	Contractor	Greece
3	Thales Communications	Thales	Contractor	France
4	ACE Associated Compiler Experts B.V.	ACE	Contractor	Netherlands
5	Compaan Design	Compaan	Contractor	Netherlands
6	Politecnico Di Torino	PoliTo	Contractor	Italy
7	ATHENA Industrial Systems Institute	Athena	Contractor	Greece
8	Universita Degli Studi Di Genova	UniGe	Contractor	Italy
9	SingularLogic	SiLo	Contractor	Greece

Document Revision History

Date	**Issue**	**Author/Editor/Contributor**	**Summary of main changes**
2012-03-30	1.0	Mihai T. Lazarescu	1st draft
2012-03-30	1.1	Luciano Lavagno	Revision
2012-04-02	1.2	Luciano Lavagno	Minor edits
2012-04-03	1.3	Ioannis Papaefstathiou	Reviewed & Approved

Table of contents

Abbreviations

FIFO First-In First-Out

IDE Integrated Development Environment

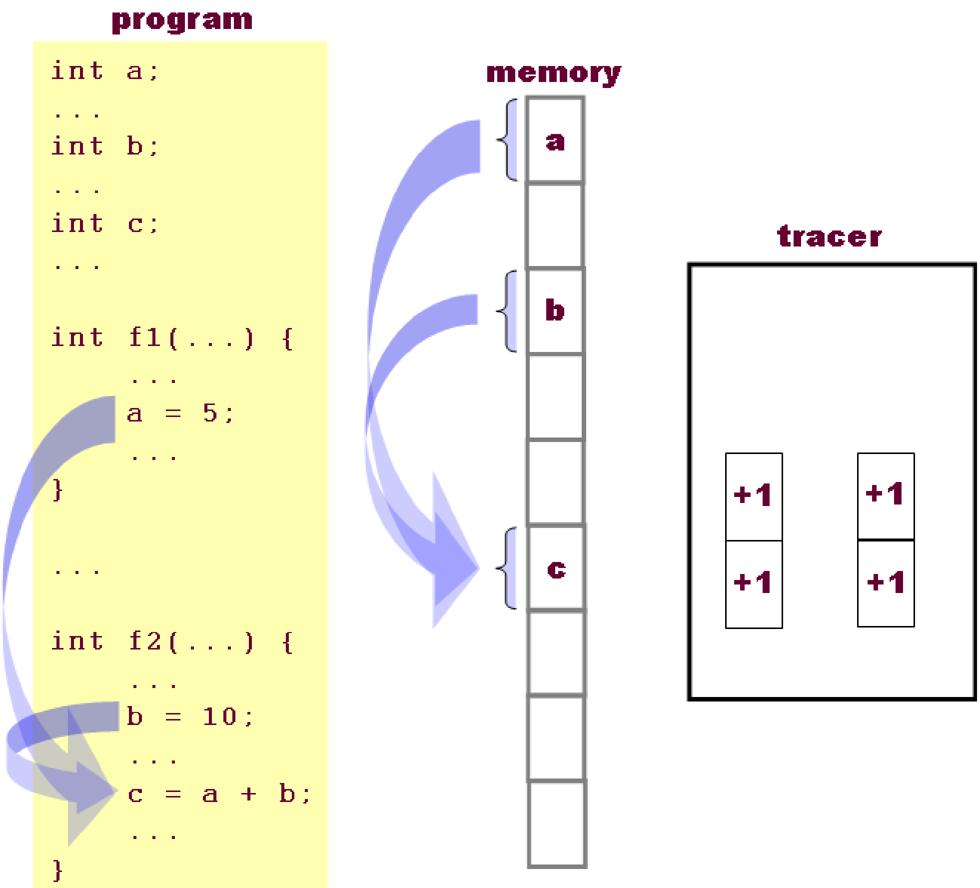
KPN Kahn Process Network

VM Virtual Machine

1. Introduction

In this section we provide an overview of the parallelization strategy used in the optimistic data dependency-based flow. This makes the report more self-contained and helps the reader to better understand the global picture. Moreover, since the parallel code generation capabilities are embedded in the same Integrated Development Environment that is used for parallelization visualization, this also makes it easier to understand its operation. The reader who is interested in more details on the operation of the first two steps of data dependency analysis and visualization is referred to D3.2 for the data dependency profiling tool and to D3.4 for the dependency visualization tool.

Basically, the parallelization flow consists of the acquisition at run time of several execution characteristics, such as the execution frequencies and data dependencies between the program instructions, as depicted below.



This is achieved by annotating the C source with data dependency profiling API calls (described in full detail in D3.1), as follows:

- `heap_enter_function(char *funcName, int sourceLine, ...)` and `heap_exit_function(char *name, int sourceLine, ...)` used to trace the call stack.
- `heap_declare(char *varName, int sourceLine, void *address, ...)` and `heap_alloc(int sourceLine, void* address, ...)` used to trace the address of static, automatic and dynamically allocated variables. For the first two categories, the name is the same as in the source code. For the latter category, the name is dynamically generated upon every execution of the memory allocation call (based on the source code line where it occurs).

- `heap_read(int sourceLine, void *address, ...)` and `heap_write(int sourceLine, void *address, ...)` used to trace at runtime the reads and writes to an address performed by a statement.

Note that the tracing technique completely solves the aliasing issue. For example, assume that the following source code:

```
1       int a, *b;
2       a = 2;
3       b = &a;
4       ... = *b;
```

is annotated as follows:

```
int a, *b;
heap_declare("a", 1, &a);
heap_declare("b", 1, &b);
a = 2;
heap_write(2, &a);
heap_read(3, &&a);
b = &a;
heap_write(3, &b);
heap_read(4, b);
... = *b;
```

The dependency is correctly identified as going from line 2 to line 4 of the original code, through variable a. Line 3 does not generate any read dependency, since &a is effectively a constant at that point of the code, and &b is not read any further in the code fragment.

The processing of the API calls at run-time results in the collection of data dependencies, where each dependency is a pair (producer statement, consumer statement) and is annotated, to help the designer reason about the code structure, with the name (and index in case of arrays) of the source variable through which the dependency occurs.

At the end of the execution, the aggregated data are displayed in an interactive graph cross-referenced with the original source code and which is used to discover and analyse the parallelism opportunities. Several examples of such graph are provided in Section . In this graph every node corresponds to a statement in the original source, and every arc corresponds to a set of addresses (labelled with the declared variable name, if applicable) written by the source node statement and read by the sink node statement.

The resulting graph can obviously be very complex, and the HEAP Graphical User Interface provides sophisticated mechanisms to:

1) collapse graph nodes at the block and function level (i.e. all the nodes belonging to a block or function become a single node, with all dependencies correspondingly accumulated).
2) accumulate dependencies into caller nodes, like the gprof tool does for execution times. In this mode, data dependencies between statements of called functions (properly uniquified based on the call tree) are attributed to the callers when the developer requests so.
3) focus on a function (as will be shown in Section 3.3) and walk over the statements that read data produced by other graph nodes

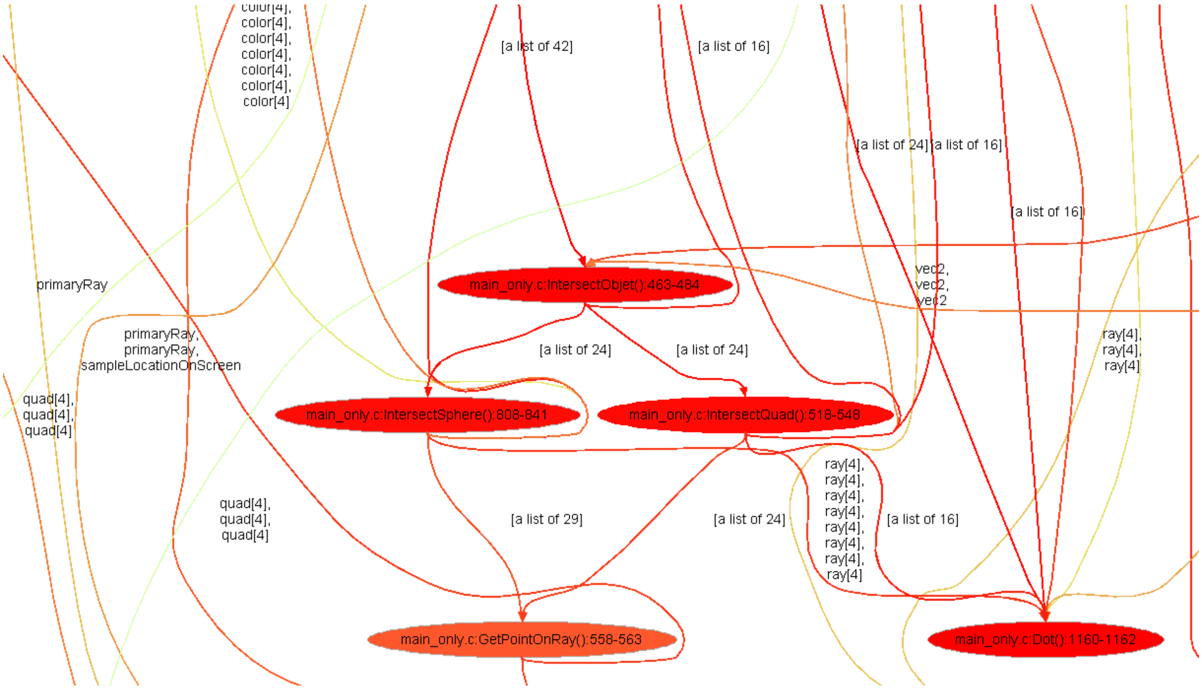
2. An example of parallelization using the HEAP tools

In this section we illustrate the use of the HEAP parallelization flow and tool suite to one of the HEAP applications from WP5, namely the ray tracer.

Ray tracing, which mimics the visual process by simulating light rays from light sources, to objects, to the eye, is a widely known “embarrassing parallel” application. However, taking an existing sequential implementation without any prior knowledge of the software, guided only by a classical source code profiler can be a daunting task. The gprof output may look like this:

```
%     cumulative self        self
time  seconds     seconds    calls      name
16.61 2.79        2.79       788215425  Dot
13.78 5.12        2.32       141631877  IntersectQuad
8.26  6.50        1.39       281277610  intersectObject
8.02  7.86        1.35       139645733  IntersectSphere
7.90  9.19        1.33       69361053   NormalizeVec3
7.69  10.48       1.29       220258108  Cross
6.42  11.56       1.08       268195824  Mul1Vec3
6.12  12.59       1.03       350638670  Sub2Vec3
4.46  13.34       0.75       45257208   IntersectionShadowWithScene
4.22  14.05       0.71       191964084  Add2Vec3
3.77  14.69       0.64       330958926  UpdateStat
3.36  15.25       0.56       15085736   CastShadowRay
...
```

however, this execution time profile does not provide any clue about how the data flows through the code. On the other hand, the output of the HEAP data dependency profiler, shown below, shows a clear uni-directional data flow through some of the functions that are at the top in the cumulative profile above, namely `intersectObject`, `intersectQuad` and `intersectSphere`.



This provides the developer with the required clue about where to focus his or her attention: loops involving these functions. These uni-directional data dependencies are essential to highlight both pipeline and doall parallelism, because they clearly identify stateless parallelizable computation.

A quick code inspection, driven by the graph above using teh HEAP GUI described in Section 3.3, shows that `intersectObject` is called exactly in two contexts, with essentially the following code structure:

```
RT_Object *obj = (RT_Object *)scene->m_firstObject;
while (obj != NULL) {
      localData.m_distance = RT_MAX_FLOAT;
      localData.m_hitFlag = 0;
      localData.m_hitObject = NULL;
      if (intersectObject(obj,ray,caster,&localData)==1)
            if (localData.m_distance < data->m_distance)
                  *data = localData;
    obj = obj->m_next;
}
...
int intersectObject(const RT_Object *obj,
                      const RT_Ray *ray,
                      const RT_Object *caster,
                      RT_IntersectionData *data) {
...
```

In this case the choice of parallelization even without any prior knowledge of the application is quite obvious. One can create a pool of worker tasks, each implementing exactly the same functionality, namely a call to `intersectObject` with `obj`, `ray`, and `caster` as inputs and `localData` as output.

Note that even though the inputs to `intersectObject` are `const` pointers, this is no guarantee a priori that they are only used as inputs, since both C and C++ notoriously allow one to cast away `const`-ness and update data structures that one could in principle consider as inputs. On the other hand, the HEAP data profiler allows one to identify precisely (within the limitation of the execution paths driven by the provided input data, of course; the reader is referred to D3.5 for more information about coverage verification) though which pointers inputs and outputs are accessed. In this case, it shows (through a more detailed inspection of the profiling data, available through the HEAP graphical user interface) that the inputs are indeed only read and the output only written.

Assuming a FIFO-based Kahn Process Network structure for parallelization, and assuming a goal of N-way parallelization, to match the parallelism of an N-way core, one could change the code above to the following form:

```
FIFO(RT_Object) objIn[N];
FIFO(RT_Ray) rayIn[N];
FIFO(RT_Caster) casterIn[N];
FIFO(RT_InterSectionData) dataOut[N];
FIFO(int) resultOut[N];
RT_Object* obj = (RT_Object*)scene->m_firstObject;
while (obj != NULL) {
      // Scatter outputs
      for (i = j = 0; obj != NULL && i < N; i++, j++) {
            objIn[i].put(*obj);
```

```
            rayIn[i].put(*ray);
            casterIn[i].put(*caster);
            obj = obj->m_next;
      }
      // Gather inputs
      for (i = 0; i < j; i++) {
            localData = dataOut[i].get();
            if (resultOut[i].get() == 1)
                  if (localData.m_distance < data->m_distance)
                        *data = localData;
      }
}
...
void intersectObjectProcess(int i) {
while (1) {
      RT_Object *obj = objIn[i].get();
      RT_Ray *ray = rayIn[i].get();
      RT_Object *caster = casterIn[i].get();
      int result;
      RT_InterSectionData localData;
      localData.m_distance = RT_MAX_FLOAT;
      localData.m_hitFlag = 0;
      localData.m_hitObject = NULL;
      result = intersectObject(obj, ray, caster, &localData);
      resultOut[i].put(result);
      dataOut[i].put(localData);
}
}
```

In the above code snippet we assume that the runtime system creates N concurrent processes, each executing the code of the function `intersectObjectProcess`, each with a different value of i from 0 to N-1.

This parallelization:

1) can be obtained very quickly. *The entire process illustrated above, including the debugging took less than two hours for a programmer with no previous knowledge of the ray tracing application*.
2) is guaranteed to be correct as long as the only communication occurs via the FIFO queues.

The latter can be observed by analyzing the data dependency information, but of course can never be guaranteed, because it may be violated along some execution paths which were not traversed due to a limitation of the input data provided to the profiler.

It is also possible to use the array-driven automated parallelization tools described in D3.3 on the same ray tracing application. parallelized the ray tracing application by choosing a different procedure, after some code rewriting in order to improve its automated parallelism discovery. The Compaan parallelization is for a loop performed over all shadow rays, while the manual parallelization is for a loop performed over

objects. Both are reasonable candidates, and the best choice depends on the relative number of iterations, which can be readily discovered by source code instrumentation.

However, it is interesting to observe that the Compaan compiler can greatly benefit from the HEAP profiler. The user of the Compaan compiler will need to do an educated guess on which part to rewrite. Typically, these are compute intensive parts which already resemble SANLP. The HEAP profiler could provide useful information on:

- where the compute intensive procedures are
- whether there are no data dependencies other than through procedure arguments
- whether procedure inputs and outputs are truly unaliased
- whether procedure inputs are truly read-only and outputs are write-only

This quick example illustrates how the HEAP approach can be used to discover multiple parallelization opportunities, leaving to the developer the choice of the one which best suits the underlying multi-core architecture. The next section will cover in details how the developed tools can be used for this task.

3. HEAP parallelization tools

A **Linux**[1] **VirtualBox**[2] virtual machine (VM) was configured to reliably support the functionality of the tool chain. Its installation is described in D3.2, to which the reader is now referred.

The users defined on the virtual machine are:

- **root** with the password: **Demo11HEAP**

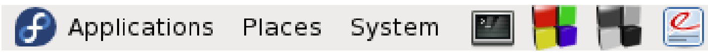
  This login can be used to perform administration tasks on the VM, if required.

- **heapdemouser** with the password: **Demo11HEAP**

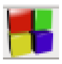
  This login is used for all tool chain-related activities.

Demo Project

The buttons to launch the applications of interest are exposed for convenience on the top panel of the workspace, right next to Fedora menus:



From left to right, they are:

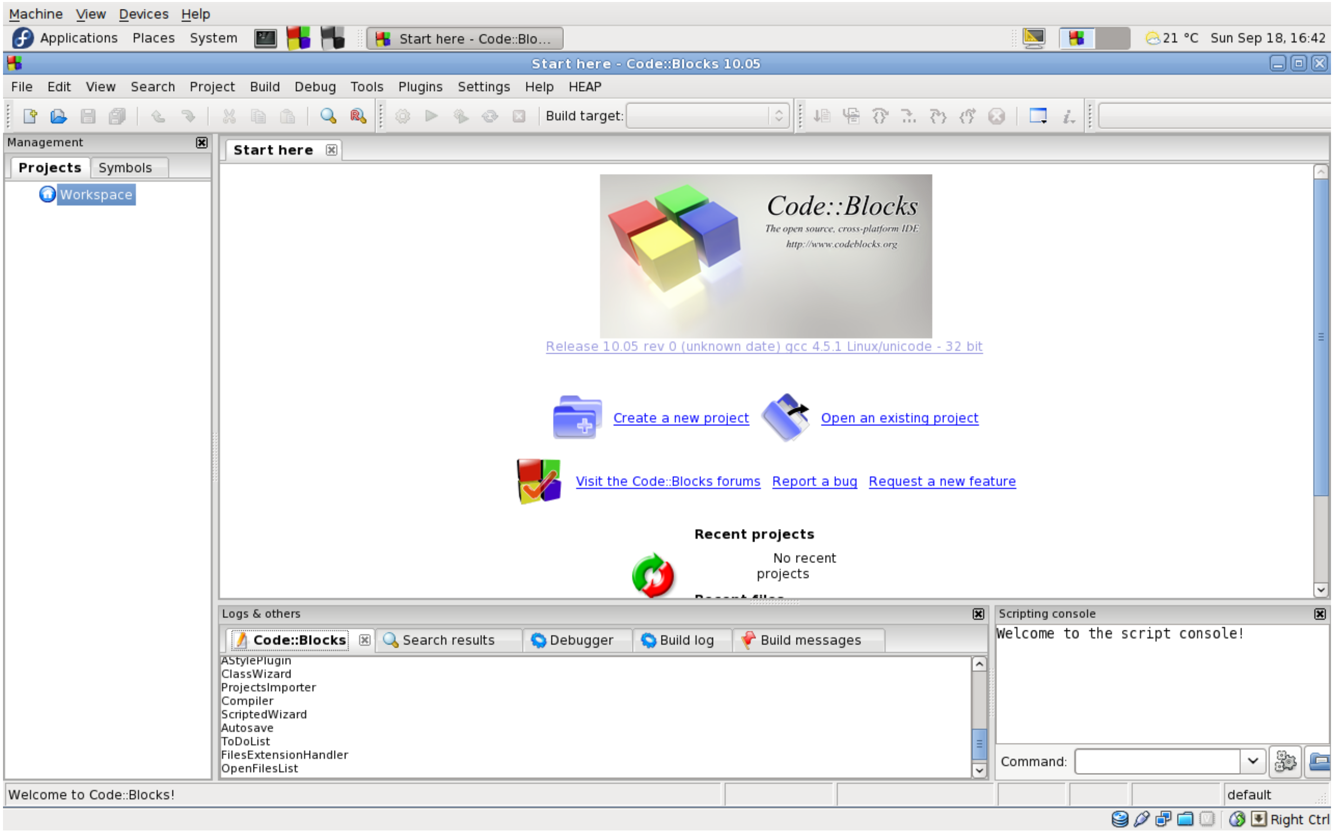
-  opens a terminal window;
-  opens Code::Blocks IDE;
-  discards a hanged instance of the Code::Blocks IDE;

[1] https://secure.wikimedia.org/wikipedia/en/wiki/Linux
[2] http://www.virtualbox.org/

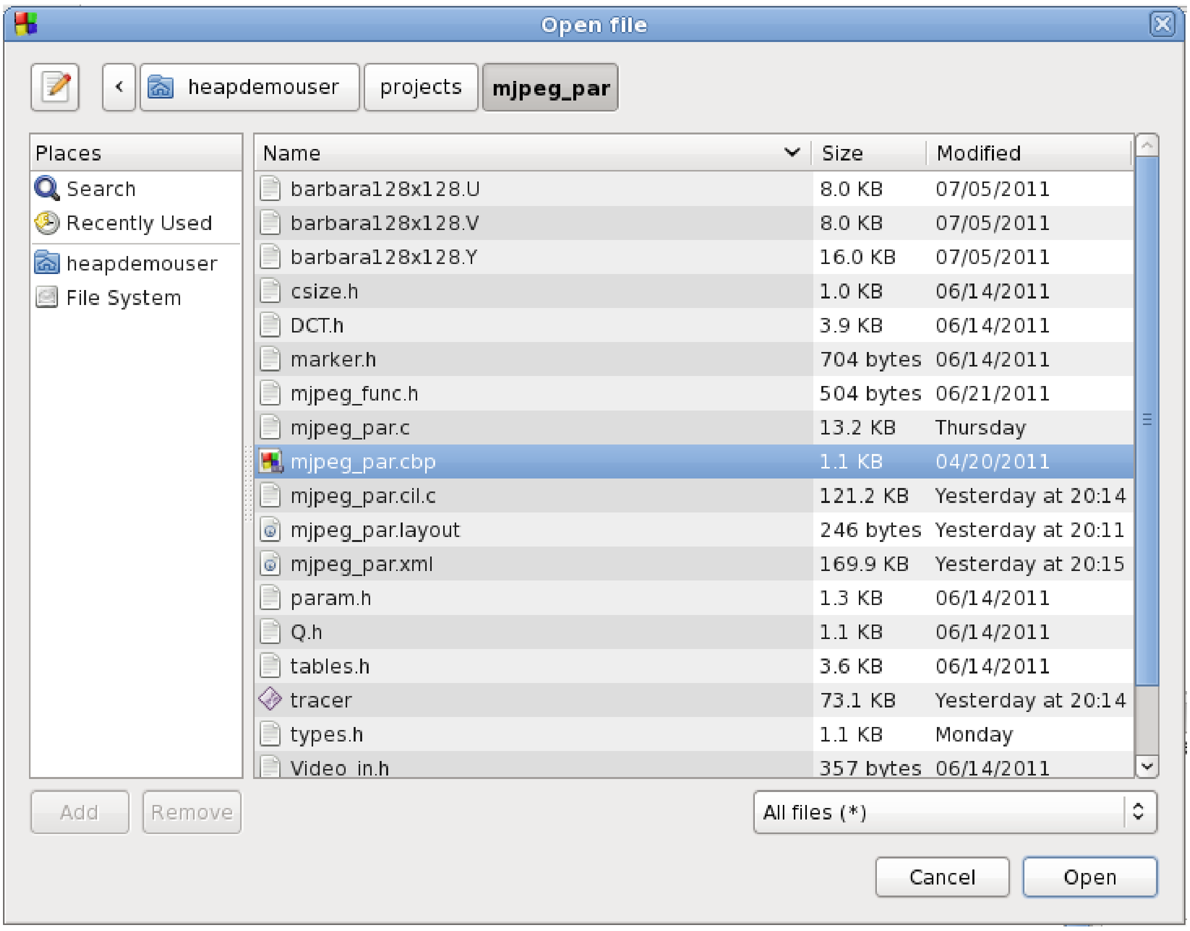
- displays the user manual of the distribution.

3.1. Load the Demo Project

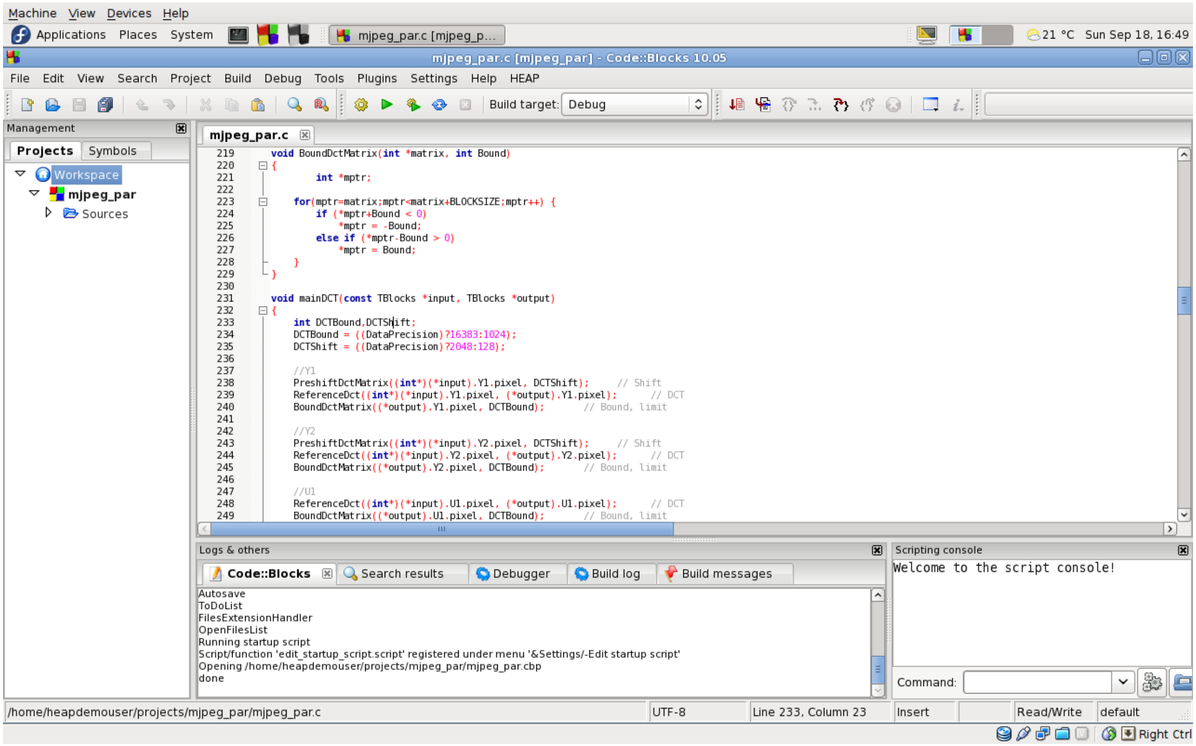
Click on the Code::Blocks button ( ) to start the IDE:



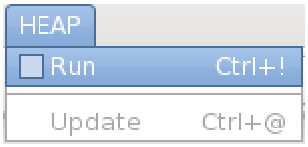
Select "**File**" from the top menu, then click on "**Open...**". In the file chooser window that opens navigate to "**heapdemouser/projects/mjpeg_par**", select "**mjpeg_par.cbp**" and click on "**Open**":



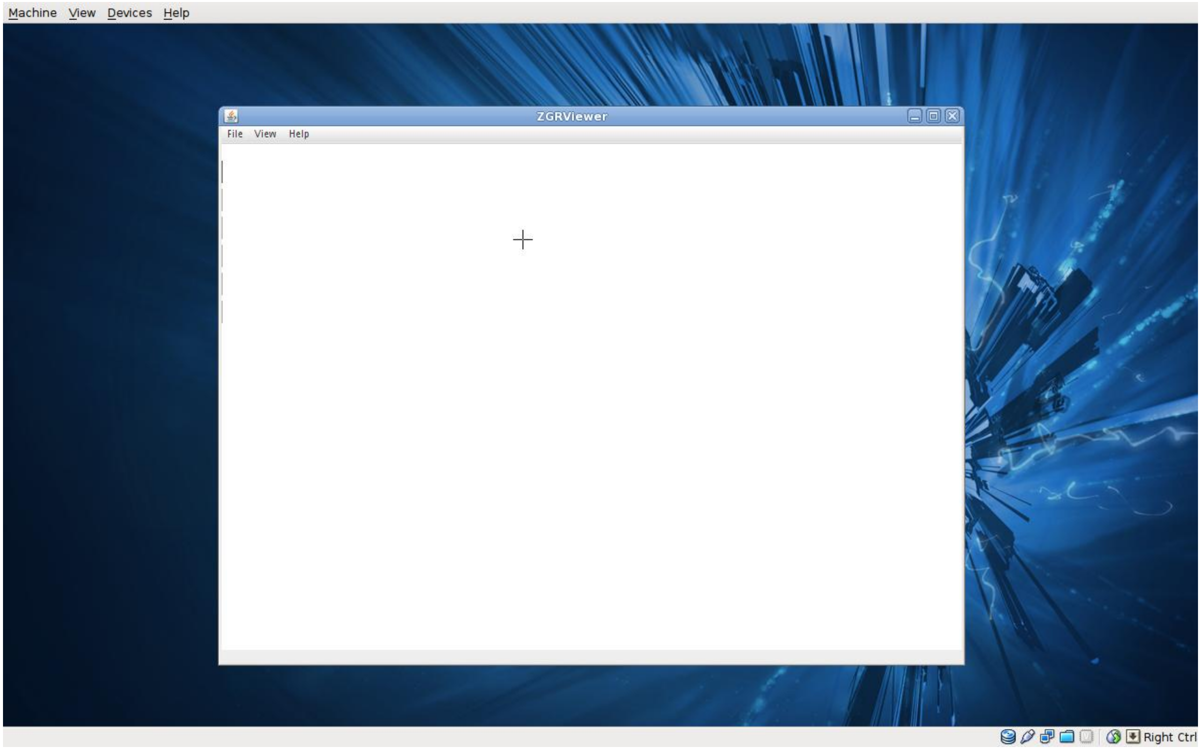
The “mjpeg_par” project will open:



Now start the visualization program by clicking on the “**HEAP**” entry of the top menu and then on “**Run**”:



The ZGRViewer visualizer window will open:



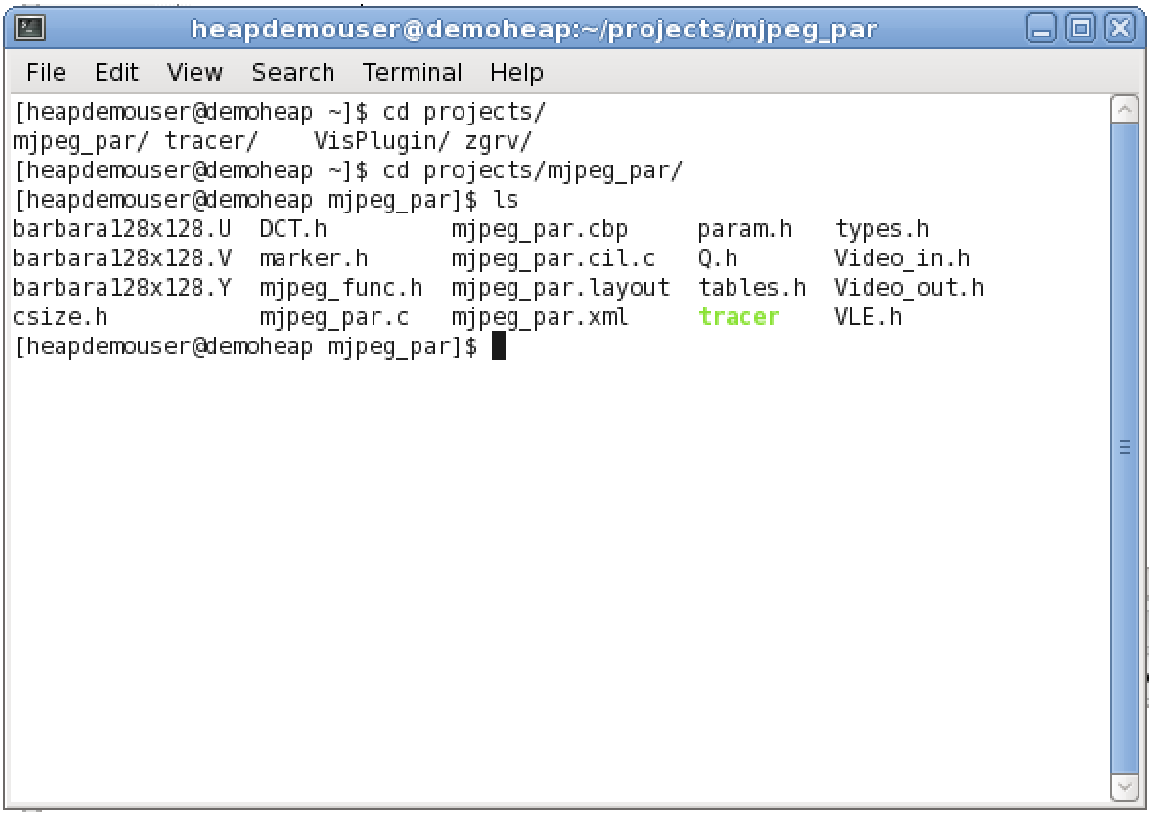
Arrange the IDE and the ZGRV windows on the screen to have a clear view of both. If you have two monitors attached to the host machine you may wish to move the ZGRV window on the second monitor of the VM and then move this VM second monitor window on the second physical monitor of the host.

3.2. Run the Demo Analysis

The analysis tool chain is run from the command line. A script is provided that loosely glues together the whole chain.

***Note**: the instrumented program runs about **450 times slower** than the native run.*

Open a terminal window by clicking on the icon in the top panel of the workspace and go into the directory of the mjpeg_par project of the IDE:



In this directory run the **tracer.sh** script with arguments:

tracer.sh -- mjpeg_par.c

where:

- -- (double dash) ends the command line options that are passed to the compiler and linker;
- **mjpeg_par.c** is the name of the source file to analyse:

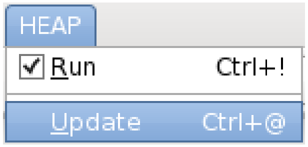
```
heapdemouser@demoheap:~/projects/mjpeg_par
File  Edit  View  Search  Terminal  Help
[heapdemouser@demoheap ~]$ cd projects/mjpeg_par/
[heapdemouser@demoheap mjpeg_par]$ ls
barbara128x128.U  DCT.h         mjpeg_par.cbp     param.h   types.h
barbara128x128.V  marker.h      mjpeg_par.cil.c   Q.h       Video_in.h
barbara128x128.Y  mjpeg_func.h  mjpeg_par.layout  tables.h  Video_out.h
csize.h           mjpeg_par.c   mjpeg_par.xml     tracer    VLE.h
[heapdemouser@demoheap mjpeg_par]$ tracer.sh -- mjpeg_par.c
+ /home/heapdemouser/projects/tracer/cil-1.4.0/bin/cilly --save-temps --noWrap -
-noPrintLn --dooneRet --dosimplify --doimarw -c mjpeg_par.c
gcc -D_GNUCC -E -DCIL=1 mjpeg_par.c -o ./mjpeg_par.i
/home/heapdemouser/projects/tracer/cil-1.4.0/obj/x86_LINUX/cilly.asm.exe --out .
/mjpeg_par.cil.c --noWrap --noPrintLn --dooneRet --dosimplify --doimarw ./mjpeg_
par.i
gcc -D_GNUCC -E ./mjpeg_par.cil.c -o ./mjpeg_par.cil.i
gcc -D_GNUCC -c -o ./mjpeg_par.o ./mjpeg_par.cil.i
+ rm -f mjpeg_par.i mjpeg_par.o mjpeg_par.cil.i
+ gcc mjpeg_par.cil.c -lavl -lxml2 -lheap -o tracer
+ rm -f mjpeg_par.cil.o
+ ./tracer
W: no arg 1 for instruction 19 (main())
W: no arg 2 for instruction 20 (main())
+ test -s model.xml
+ test model.xml = mjpeg_par.xml
+ mv model.xml mjpeg_par.xml
[heapdemouser@demoheap mjpeg_par]$
```

where:

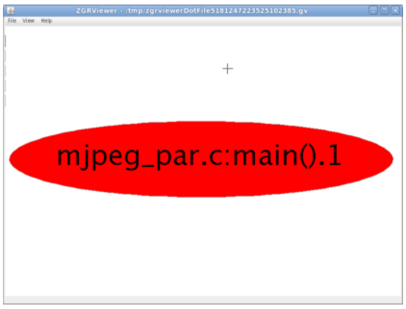
- **+ /home/heapdemouser/projects/tracer/cil-1.4.0/bin/cilly** is the starting command for CIL compilation;
- the three **gcc** compilations that follow are part of the cilly run and generate the instrumented model of the user program, **mjpeg_par.cil.c**;
- **+ rm -f mjpeg_par.i mjpeg_par.o mjpeg_par.cil.i** cleans the temporary files form the directory;
- the next **gcc** run compiles the CIL model (mjpeg_par.cil.c) and links it with the data dependency tracer library (libheap) and other system libraries (libxml2, libavl);
- the **rm** command cleans the temporary files from the directory;
- then the data dependency tracer is run. It actually runs the user program instrumented for data dependency tracing together with the data dependency tracer;
- finally, the **mv** command renames the file with the generated data to the name expected by the ZGRViewer-based visualizer.

3.3. Run the Data Dependency Visualization

After each operation that can affect the visualization (e.g., an update of the visualizer data) the visualizer should be informed on the update. Access the HEAP menu on the top menu of the IDE and click on the “**Update**” entry:



The visualizer window displays the start view of the program execution in its most compacted form: the main() function completely collapsed in a single node.



The ellipse represents the function where the program execution starts, the main() function. All its instructions, their data dependencies, called functions and their data dependencies are recursively collapsed into the most synthetic representation of the program execution.

The collapsed nodes are named using the following convention:

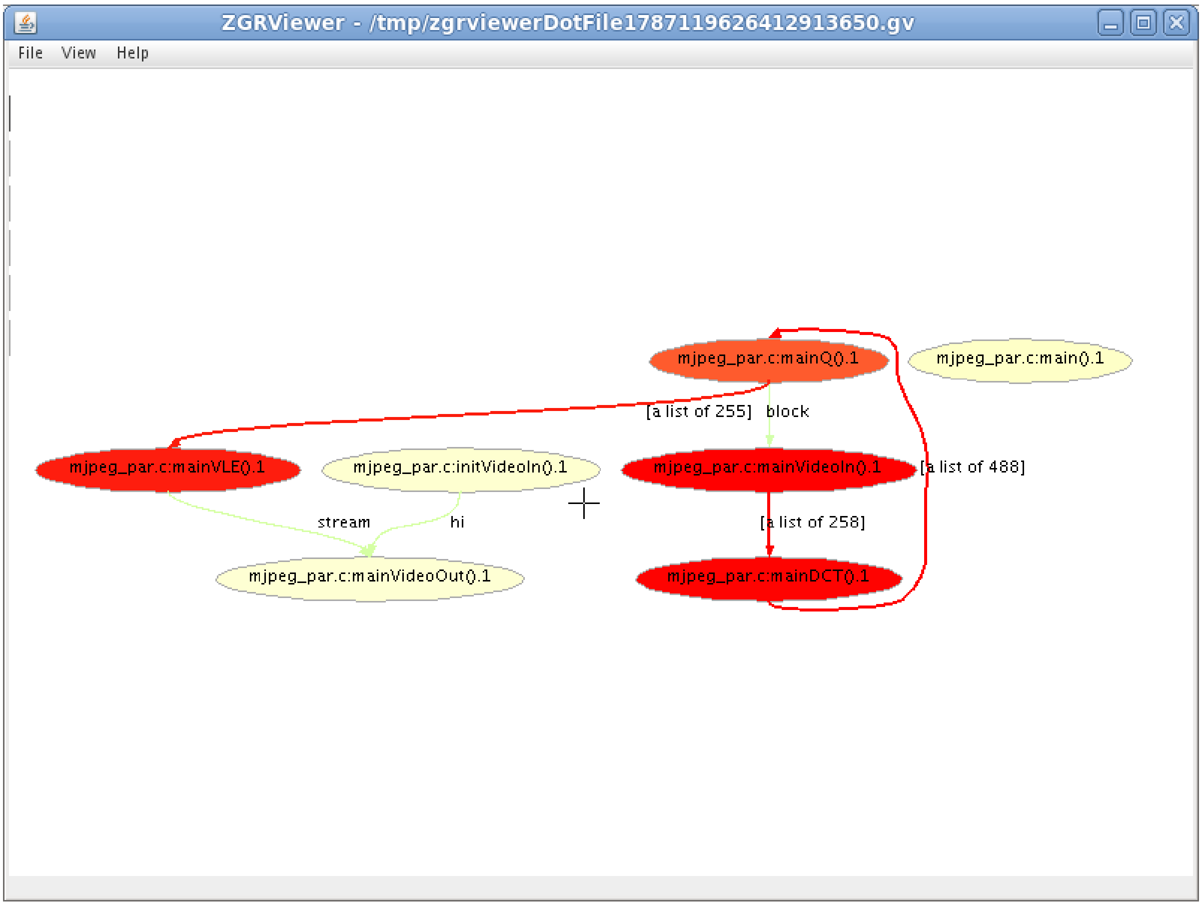
- the source file name and path relative to the root of the project. The root of the project is automatically calculated as the longest common part of the path of all the source files of the project. In the example in the figure above: 'mjpeg_par.c';
- a separator, ':';
- the C function name followed by an open and closed parenthesis, '()'. In the example in the figure above 'main()';
- a separator, '.';
- the call stack ID as an integer. In the example in the figure above '1'.

Such a synthetic view of the program is of little use. It is meant to allow the developer to start the analysis of the program from the most logical spot and proceed to the exploration of the execution data by following the most promising path for parallelisation.

The contents of each collapsed node can be expanded in two successive stages:

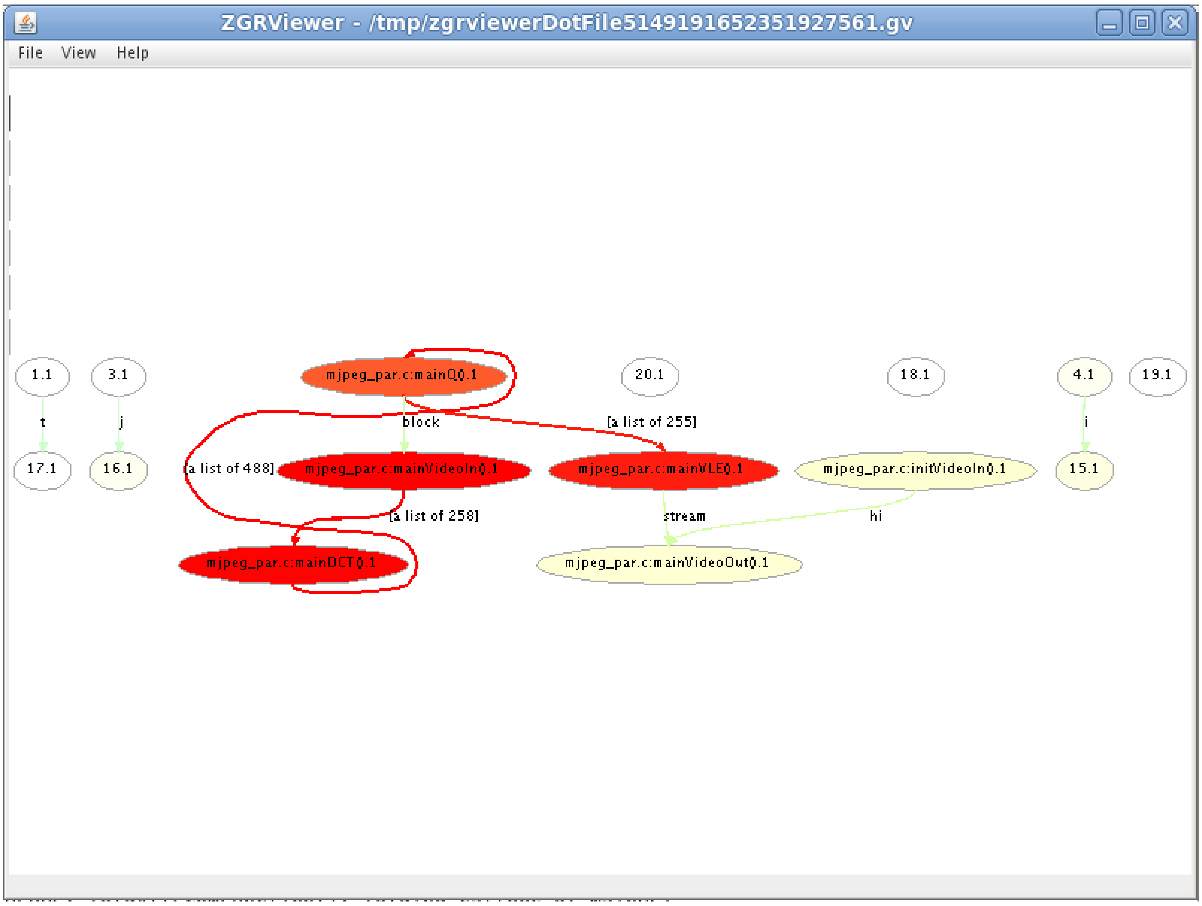
1. the first expansion is *at function level*. It expands the contents of the collapsed node up to the functions that it calls directly. The other instruction of the node are not expanded at this stage, thus the collapsed node will be still displayed as it still holds the collapsed instructions.
2. the second expansion is *at full node level*. This time the collapsed node is fully expanded and will be replaced completely in the view by all its nodes and functions that it calls directly.

The levels of expansion of the starting main() function are depicted below. The function-level expansion:



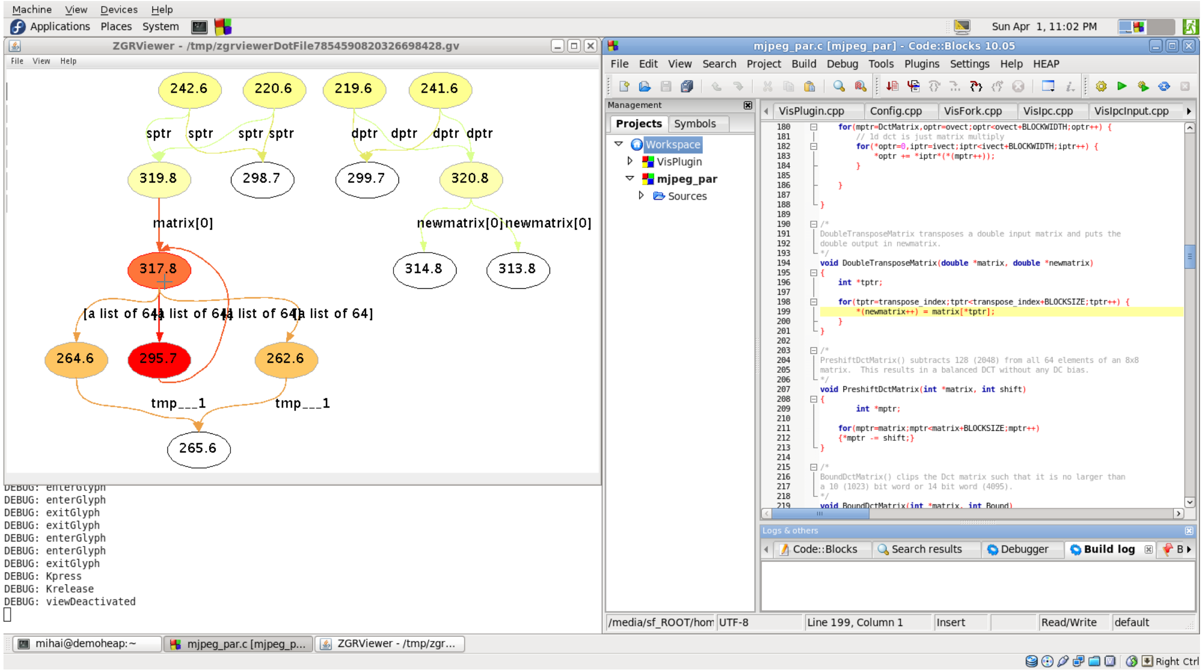
exposes all functions called by the main() function of the program and the data dependencies among them. The main() function is still collapsed (top-right ellipse in the figure) as it still holds the nodes for the instructions in the main() function of the program under analysis.

The second level of expansion of the main() function is the full node expansion:



At his level, the folded node corresponding to the main() function is completely expanded. Being void of contents it is and no longer represented on the graph. Its contents is represented expanded into the nodes corresponding to its instructions and to folded nodes corresponding to the functions called by the main() function, that recursively fold their contents and that of the functions they call in the various call stacks.

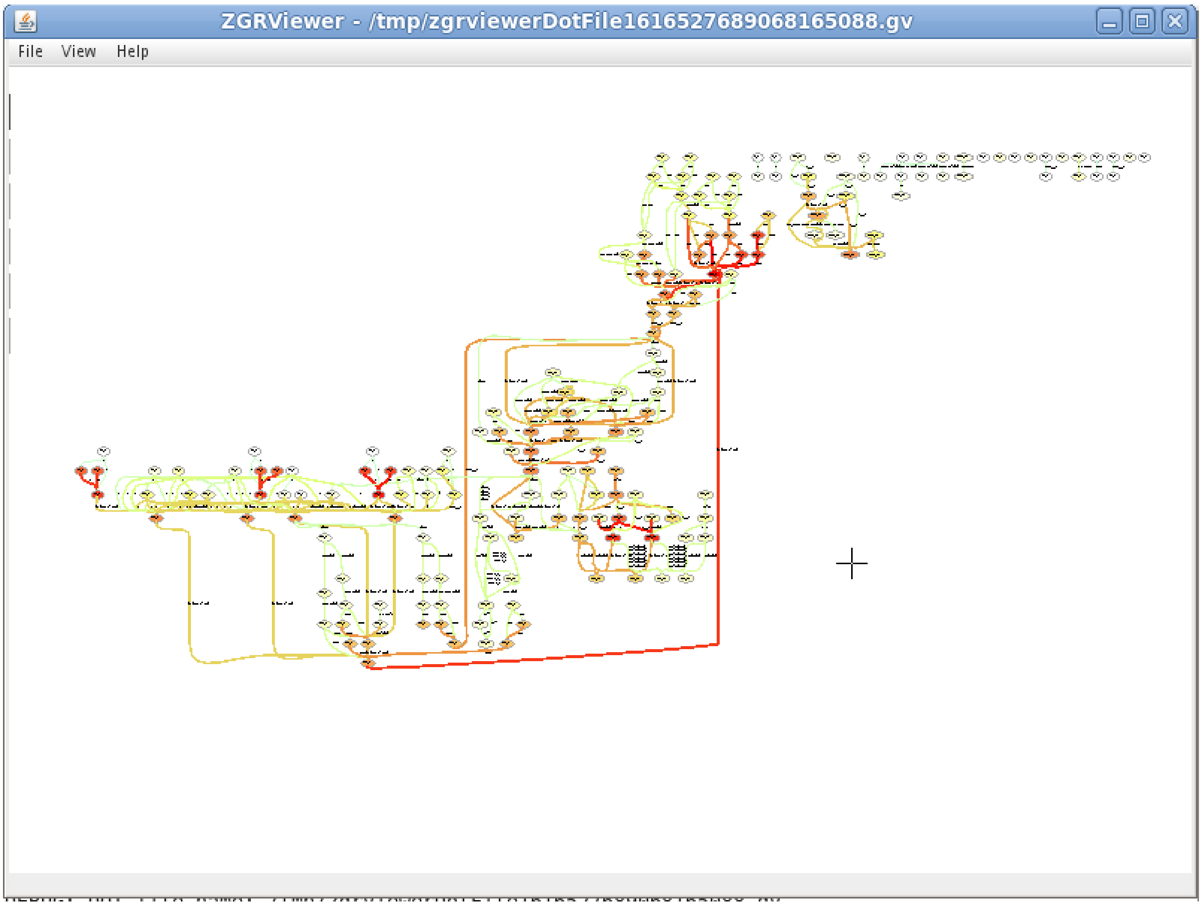
Of particular importance is the capability to display only the nodes that handle data exchanged over a given function boundaries. The following figure exemplify this filter applied to the 'DoubleTransposeMatrix()' function:



For instance, node 317 in call stack 8 (317.8) correspond to the highlighted line in the source file in the IDE and its data dependencies across function boundaries with the node 295 in call stack 7 (295.7), that can be quickly located using the Viewer-IDE cross-reference functions as belonging to the innermost loop of the function 'DoubleReferenceDct1D()' are clearly emphasized with the usual colour encoding for their importance during the program execution.

These dependencies represent candidates for inserting communication FIFOs if the two functions are to be extracted into parallel execution kernels.

The progressive expansion of the nodes and data dependency analysis across function boundaries helps the developer focus the exploration for parallelisation opportunities by hiding the complexity of the parts of the program that are not of interest at any given time. This is very important, as the program data dependencies, when represented to their fullest extension, become quickly too complex to be analysed even for relatively simple programs:

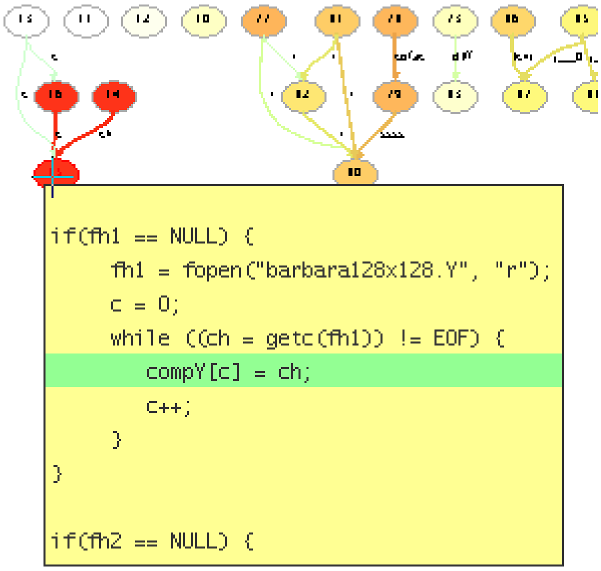


Each ellipse represents a program instruction that was executed. The ellipse colour can go from white (seldom executed) to intense red (most executed).

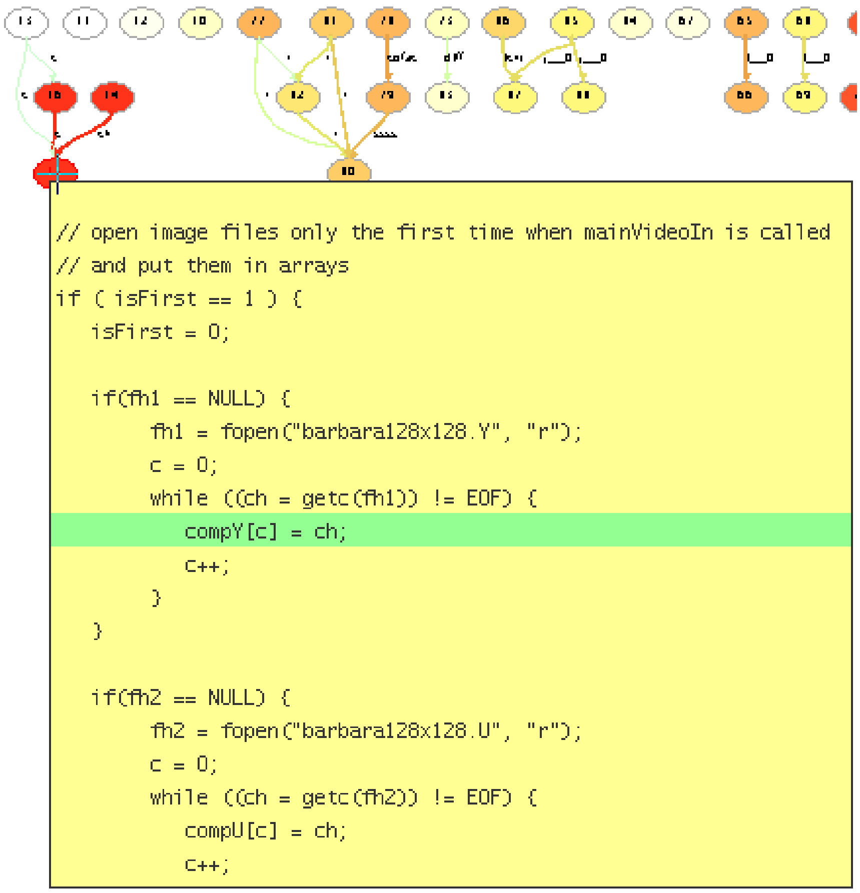
Each directed arch that connects two ellipses represents a data dependency between the two instructions. The arch colour can go from light cyan (for seldom occurring dependencies) to intense red (for most occurring dependencies) and, at the same time, also the arch width is modulated by the same factor, the widest for the most occurring.

The visualizer implements a few handy short cuts:

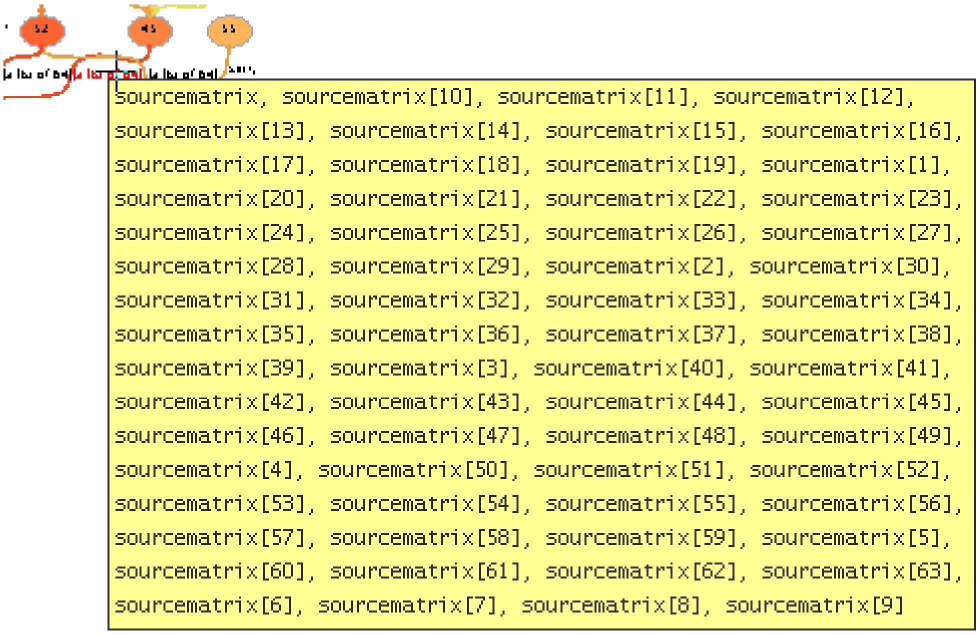
- ‘**c**’ -- with the cursor on a node, display the source code of the node with 5 context lines:



- ‘**C**’ -- with the cursor on a node, display the source code of the node with 10 context lines:



- ‘**e**’ -- with the cursor on a node, move the IDE editor cursor on the source line corresponding to the node;
- ‘**m**’ -- with the cursor on an arch, display the unabridged list of data dependencies represented by the arch:



- **‘g’ --** best fit of the graph on screen;
- **mouse wheel** is used the to zoom in/out;
- **click and drag** is used for panning in any direction.

4. Conclusion

This document illustrates the use of the HEAP toolset for parallelization based on the results of run-time data dependency analysis. It contains both an overview of the flow and a detailed explanation of the steps involved. Comments and suggestions from HEAP partners are welcome.