POLITECNICO DI TORINO
Repository ISTITUZIONALE

HEAP: A Highly Efficient Adaptive multi-Processor framework

(Article begins on next page)

04 August 2020

HEAP: a Highly Efficient Adaptive multi-Processor framework

Luciano Lavagno[1], Mihai T. Lazarescu[1,], Ioannis Papaefstathiou[2], Andreas Brokalakis[2],
Johan Walters[3], Bart Kienhuis[3], Florian Schäfer[4]

[1]Department of Electronics, Politecnico di Torino, Torino, Italy
[2]Synelixis Solutions Ltd, Greece
[3]Compaan Design, Netherlands
[4]FS Result GmbH, Germany

Abstract

Writing parallel code is difficult, especially when starting from a sequential reference implementation. Our research efforts, as demonstrated in this paper, face this challenge directly by providing an innovative toolset that helps software developers profile and parallelize an existing sequential implementation, by exploiting top-level pipeline-style parallelism. The innovation of our approach is based on the facts that a) we use both automatic and profiling-driven estimates of the available parallelism, b) we refine those estimates using metric-driven verification techniques, and c) we support dynamic recovery of excessively optimistic parallelization. The proposed toolset has been utilized to find an efficient parallel code organization for a number of real-world representative applications, and a version of the toolset is provided in an open-source manner.

Keywords: software parallelization, parallelization tools, trace-based data dependency analysis, data dependency analysis, parallelization verification, automatic parallelization

1. Introduction

Writing parallel programs has traditionally been considered a difficult task, even when parallelism is taken into account from the beginning. Moreover there is an urgent need to parallelize the massive amounts of legacy sequential code so as to increase its performance on processors and systems that refocus from single-thread acceleration to increasing the overall throughput. Automated software parallelization has been tackled extensively at the instruction level and loop level, which are appropriate for VLIW and vector processors. However, only some past work, namely the Compaan approach [1], has actively sought parallelization opportunities at the *task* level, which are most appropriate for modern multi-core processors. A main limitation of the latter techniques is that so far they required loops enclosing the main computational bottlenecks to have very simple internal control (no early exit, limited support for conditionals, etc.), and data access patterns (very limited use of pointers, only affine array indices, etc.)

The HEAP project faces these challenges directly, by developing an innovative toolset that helps software developers profile and parallelize existing sequential implementations by exploiting top-level pipeline-style parallelism. It synergistically uses and extends with respect to past work:

1. The above mentioned Compaan approach [1], [2], extended to support a significantly larger set of control structures (as detailed in Section 3), to provide automatic parallelization capabilities based on full *compile-time* dataflow analysis on a *reduced scope* of the application with a *reduced complexity* (in HEAP referred to as the *pessimistic* approach).
2. A novel approach, related to [3] and [4] and described in Section 4, which uses *run-time, full scope* data-dependency tracing and sophisticated graph visualization techniques to enable the code developer to *optimistically* find the best *manual parallelization opportunities*.
3. Coverage analysis and runtime tracing, as described in Section 5, to help the developer verify the manually or automatically parallelized code.

The HEAP project also covers the multi-core computer architecture side, by providing innovative cache coherency strategies, described in [5]. They exploit the data access information provided by the parallelization tools mentioned above to provide improved performance at a dramatically reduced cost with respect to current directory-based methods.

The key observation that allows such an improvement is that explicit classification (e.g., by means of a specific encoding of some bits of the address) of each load or store as operating on private or shared data leads to effective use of a write-through policy for shared data, and of a write-back policy for private data. Since all communication in the parallel model of

computation considered in this paper, namely Kahn Process Networks (KPNs [6]), is via single-reader single-writer FIFOs, this classification can be readily performed. In particular, all local process variables are private, while all FIFO variables (indices and buffers) are shared. Since most of the data accesses are on private data, the more efficient write-back policy can be applied for a majority of the accesses, thus providing the claimed cache performance and cost gains.

Note that the HEAP approach is *not limited to a KPN-style parallelism*. Only the automated parallelization path uses it, while manual parallelization (helped by the tools described in the rest of this paper) can use any style of parallel code writing. In this paper, we will consider KPNs because they are intuitive and, thanks to their deterministic behavior independent of the execution timing and schedule, they make manual parallelization easier and less error-prone.

2. Related Work

Code parallelization is one of the most widely studied topics in compilers for parallel machines since the 1970's. Most of the work so far has focused on the identification of code sections within innermost loops (Fortran "do" or "for" and "while" in C) which can be executed fully in parallel ("do-all") due to the lack of dependencies, or on a vector machine, or as a software pipeline [7], [8], [9]. However, the level of parallelism that can be identified using these techniques is very limited, since it can be significant only for special applications (physics, fluid dynamics, structure engineering), and cannot fully exploit the current architectures dominantly used for gaming and multi-media applications.

On the other hand, there is a strong need for techniques which can help the developer to manually partition an application, going beyond the limitations of automated analysis. For example, we would like to work at the "main" program level (as opposed to the innermost loop level) [10], [11].

One of the most effective automated approaches so far, which is part of the background research of this proposal and will provide us with the "lower bound" to the amount of parallelism, is the Compaan project run at the University of Leiden [1] which is being commercialized by Compaan Design. The Compaan approach focuses on Static Affine Nested Loop Programs (SANLP), which use affine loop bounds and index expressions on non-aliased data. Originally the technology was based on Matlab syntax as input specification and currently it is based on ISO C.

A similar approach, but oriented to the discovery of parallelism in the outermost loops, is used by the PICO high-level synthesis software, currently commercialized by Synopsys [11].

Another similar technique, proposed recently by Amarasinghe and others, provides the basic idea of the "upper bound" to the discovery of parallelism [12]. In HEAP, we have extended it by providing further techniques, based on data compression and advanced visualization, to show the very large amount of data that can be provided by a full data trace of, e.g., a large video encoding or decoding technique.

Several compilation and debugging tools, often based on proprietary extensions of the C language, have also been proposed by dominant industrial players. For example, Apple introduced recently the Grand Central technology based on OpenCL, a newly developed programming language. Its potential scope is parallelization of C-like programming languages for execution on graphics processors. NVidia proposed the CUDA language, very similar to OpenCL, which can be used to translate sequential C code into parallel threads that can be run on NVidia's GPUs and Stream is AMD's similar offering.

The recently announced Prism tool from criticalBlue tackles the same problem of legacy sequential code parallelization. It essentially predicts the application performance under different thread decompositions, and the corresponding inter-thread dependencies. Like in our case, its assessment of parallelization opportunities is bound by the quality of the testbench used.

3. Static array-based data dependency analysis

This section provides a short overview of how the Compaan compiler [1] performs its data dependency analysis and identifies sections of code which can be safely executed as concurrent Kahn Process Network (KPN [6]). In KPNs, processes are allowed to communicate only via single-reader single-writer FIFO queues with blocking read semantics, thus ensuring deterministic behavior by construction, regardless of the execution timing.

Compaan's exact dataflow analysis operates at the procedure level (reduced scope) and performs its analysis on C code that adheres to the reduced complexity of Static Affine Nested Loop Programs (SANLP). The Compaan compiler converts a SANLP into an efficiently pipelined KPN. The more repetitive the original program, the more effective the Compaan approach is. Especially applications in the domain of video, telecom and imaging can be easily fit in the SANLP format. This provides a productive approach to convert these applications into multithreaded, streaming implementations.

Within SANLP, control flow decisions and data access patterns depend on compile-time known values, i.e., static affine expressions. Therefore, a SANLP comprises the following:

- C loops equivalent to the form `for (it = e1; it < e2; it += e3) {...}` where `it` is the integer iterator, `e1`, `e2` are static affine (loop invariant) expressions and `e3` is a constant.

- if-then-else statements in the form if([!] e1 [<, <=, ==, >=, >, |, &] e2) {…} else {…} where e1, e2 are static affine expressions, i.e., all boolean expressions based on static affine values.
- Statements that read, write and compute locally declared unaliased scalars s and/or unaliased multi-dimensional arrays a[ ], b[ ][ ], c[ ][ ][ ]… indexed with expressions that are static affine. A statement can consist of procedure calls and standard unary or binary C operators.

Expressions of the form: $c_0*it_0+c_1*it_1... +c_i*it_i+c$ are static affine if c_0, c_1, …, c_i, c are constants and it_0, it_1, …, it_i are one of the following:

- Static affine nested loop iterators
- Run-time constants
- One of the following pseudo-linear expressions, where *s* is static affine and *c* is constant:
  - s%c
  - s/e
  - *div_floor*(s, c)
  - *div_ceil*(s, c)
  - *max*(s, s)
  - *min*(s, s)

All code *outside* the procedures analyzed by the Compaan compiler does not need to be SANLP. Data dependencies between procedures *called* by the SANLP need to be explicit through their function arguments, rather than sharing any global data.

For example, let us consider the following SANL program:

```
int a[10], b[10][10];
for (int i = 0; i < 10; i++) {
  a[i] = Function1();
  for (int j = i; j < 10; j++) {
    b[i][j] = Function2(a[i]);
  }
}
```

This example shows first the definition of the array variables a[ ] and b[ ][ ]. This is followed by two nested loops with two enclosed function calls. The for-loops define the loop iterators i and j. The function call Function1 is placed before the inner for-loop, which results in a non-perfect nested loop. Compaan can handle such non-perfect nested loops. In the example, all exchanges of data between the function calls are through the arrays a[ ] and b[ ][ ]. The indexing of the arrays is expressed in linear combinations of the loop iterators i and j. Actual computations are hidden by the functions Function1() and Function2().

Given a SANLP, Compaan can analyze the data dependencies between any pair of statements using the theory described in [2], and also show them graphically using its GUI, as shown in Figure 1.


```
#pragma compaan_procedure accumulator2d
void accumulator2d(short data_in[MAX_I][MAX_J], // 16 bit
        int data_out[MAX_J]) { // 32 bit
    int i;
    int j;
    short a[MAX_I][MAX_J];
    int sum[MAX_J]; // Partial sum

    // Initialize the partial sum
    for (j = 0; j < MAX_J; j = j + 1) {
        sum[j] = 0;
    }

    // Stream in data_in and accumulate
    for (i = 0; i < MAX_I; i = i + 1) {
        for (j = 0; j < MAX_J; j = j + 1) {
            a[i][j] = data_in[i][j];
            accumula E0(a[i][j], sum[j], &sum[j]);
        }
    }

    // Copy the last partial sum and stream out
    for (j = 0; j < MAX_J; j = j + 1) {
        data_out[j] = sum[j];
    }
}
```

Figure 1. Data dependencies overlayed on source code by Compaan GUI

For example, given the following code:

```
void accumulator2d(
   short data_in[MAX_I][MAX_J],
   int data_out[MAX_J])
{
   int i, j;
   short a[MAX_I][MAX_J];
   int sum[MAX_J];           // Partial sum

   // Initialize the sum array
   for (j = 0; j < MAX_J; j = j + 1) {
      sum[j] = 0;
   }

   // Stream in data_in and accumulate
   for (i = 0; i < MAX_I; i = i + 1) {
      for (j = 0; j < MAX_J; j = j + 1) {
         a[i][j] = data_in[i][j];
         accumula  (a[i][j],sum[j],&sum[j]);
      }
   }

   // Copy the partial sums and stream out
   for (j = 0; j < MAX_J; j = j + 1) {
      data_out[j] = sum[j];
   }
}
```

Compaan derives for this single-threaded, global memory code the KPN described in Figure 2. Each vertex in the graph represents a statement in the source code and will be implemented as a separate thread. The data dependencies (expressed as edges) are converted into FIFO communication channels. The Compaan compiler automatically produces a KPN for each SANLP and implements the KPN on a multithreaded environment based on pthreads or Intel Thread Building Blocks (TBB).
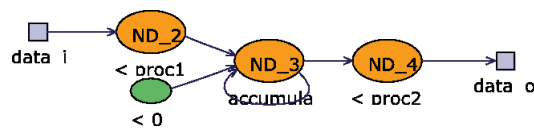

Figure 2. Example of KPN for the `accumulator2d` *function*

4.   Dynamic trace-based data dependency analysis

Static analysis techniques, as argued in the previous section, can help the developer automatically parallelize data-intensive code, with limited support for control structures or memory access modes beyond affine indices within uniform vectors. While many embedded applications fall into this category, there is a large amount of legacy software which includes a significant amount of control and decisions, or uses pointers and dynamic memory allocation intensively.

The HEAP optimistic software parallelization toolset has been developed specifically to address this second class of applications. It can be applied to any sequential C-language code and helps the software developers to profile and parallelize it by exploiting top-level pipeline-style parallelism.

The parallelized code considered in this paper (only for the sake of easier illustration, as mentioned above) uses the KPN model of computation as the code produced automatically by the tools described in Section 3. This model ensures deterministic behavior with arbitrary parallel process execution times, i.e., completely avoiding data races, in order to ease the verification task discussed in Section 5. Note that even though in general the deadlock-free executability of a KPN model in finite memory is undecidable, a KPN derived from the parallelization of an existing reference sequential implementation is guaranteed to be schedulable.

The sequential code is manually split (as illustrated in Section 6) into multiple sequential processes that are assigned to parallel resources of the architecture and use FIFO channels for inter-process communication.

The HEAP toolset flow, shown in Figure 3 is divided in four stages: (I) source instrumentation, (II) runtime trace collection and compaction, (III) trace data visualization and analysis, and (IV) manual source code parallelization. Each stage is driven by one or more of the toolset components: the CIL-based C source annotator, the execution tracer library, the ZGR viewer-based trace data graphical visualizer, and an IDE for project development, toolset integration and cross-reference between the visualizer and the source code.
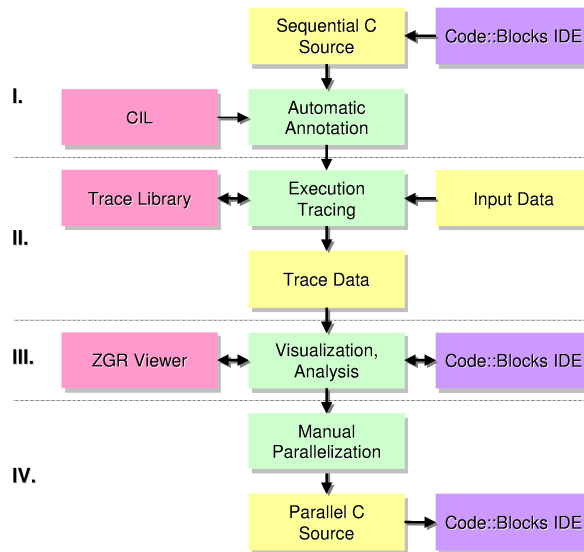

Figure 3. The HEAP optimistic software parallelization toolset flow.

In the first stage, the CIL-based [13] source code annotator automatically analyzes and rewrites the original C source to add the run-time calls to the tracer library API that are needed for a proper tracing at run-time of the program execution and the data dependencies.

In the next stage, the instrumented program is compiled, linked with the tracer library and executed using an input data set provided by the developer. The data set should be selected to *maximize the discovered dynamic data dependencies* by exercising as many statement-to-statement data dependencies as possible. The execution data are automatically collected and compacted by the tracer library functions during the run and saved at the end for display and analysis in the following stage.

Basically, the tool operation consists of the acquisition at run time of several execution data, such as the execution frequencies and data dependencies between program instructions, as shown in Figure 4.

```
a = b + c;   /* instruction ID: 10 */
x = y + z;   /* instruction ID: 100 */
r = x + a;   /* instruction ID: 1000 */
```
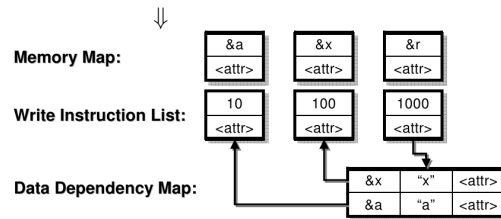
⇓

Memory Map:

&a		&x		&r
\<attr\>		\<attr\>		\<attr\>

Write Instruction List:

10		100		1000
\<attr\>		\<attr\>		\<attr\>

Data Dependency Map:

&x	"x"	\<attr\>
&a	"a"	\<attr\>

Figure 4. Basic operation of the data dependency tracing tool.

This is achieved by annotating the C source with data dependency profiling API calls, 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, …)` that are 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, …)` that are used to trace at run time 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[1]:

```
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, each linking a pair of *(producer_statement, consumer_statement)* as shown in Figure 4. The link is annotated with the name (and index in case of arrays) of the source variable through which the dependency occurs, to help the designer reason in terms of source names.

The large amount of trace data and execution statistics collected are displayed as a graph in the third stage of the HEAP toolset flow (see Figure 5). The format and the tools provided facilitate the designer search for parallelization opportunities,

---

[1]   For the sake of simplicity we consider a very simple source line identification mechanism here. In reality, the annotator uses both the line number and the source file name to identify source code locations.

e.g., pairs of statements or functions with uni-directional data dependencies that can be executed in parallel as stages of a coarse-grained task pipeline.

Each node of the graph corresponds to data processing in the original source (a statement, a function, or a collapsed call stack), and every arc corresponds to a set of addresses (labeled with the declared variable name, if applicable) written by the source node and read by the sink node.

The full data dependency graph is usually very complex. To simplify its exploration and analysis, the HEAP toolset Graphical User Interface provides sophisticated mechanisms, such as:

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). Figure 6 shows an example of function-level collapsing.
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 6) and walk over the statements that read data produced by other graph nodes and write data consumed by other graph nodes.
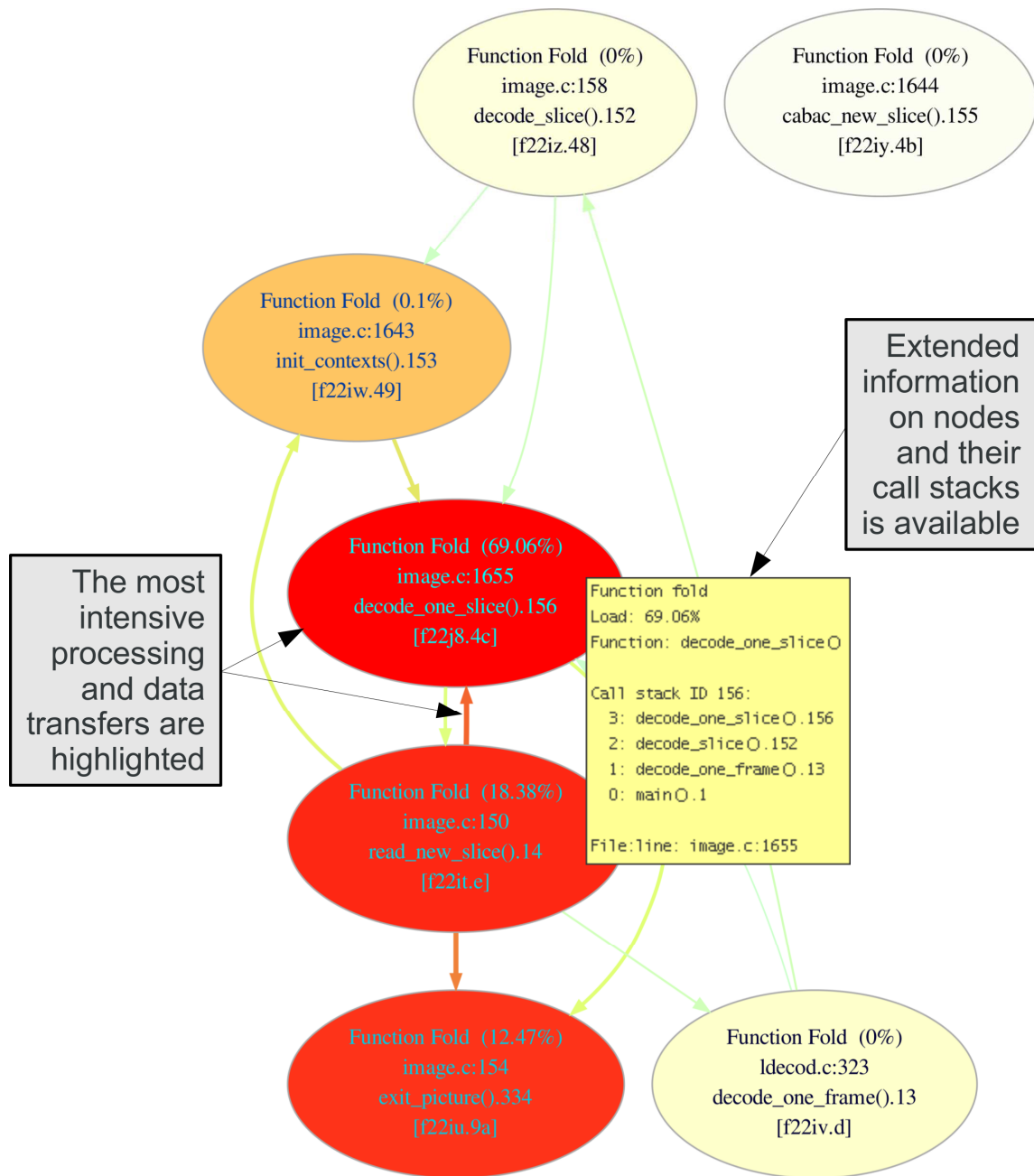
Figure 5. The graph displays program processing elements as nodes and their data dependencies as directed edges. Extended information can be obtained for each element.

5. Parallelization verification

The methodology used for verification of the optimistic parallelization described above is based on annotations added to both the initial sequential ("golden") version of the program and to the automatically or manually parallelized version. Note that also in case of automated parallelization, which is correct by construction, verification is desirable in order to discover and correct tool bugs.

The annotations produce at runtime a log file that contains data about the program execution that is then analyzed by the analysis tool. The annotation statements are provided by a verification API library. The places where a parallelization tool (or a human developer) changes the code to split the program into multiple parallel sections are the same places where the annotation API calls need to be added in order to track the program state before and after a parallel section. Consequently, the information required to parallelize a program is sufficient to also add the annotations.

Coverage analysis is performed by checking whether every checkpoint in the program has been encountered. If a checkpoint is encountered, the surrounding code has been executed. By placing a checkpoint in every branch of the code, this allows to verify that all branches have been executed. In order to perform that check, the analysis tool requires the checkpoint API calls in the program code, a structure file giving the analyzer a list of all checkpoints and possibly multiple resulting log files to check. The latter might be required because depending on the program structure it might not be possible to visit every branch of a program with a single run -- especially if error conditions are to be checked. In this case, a successful standard run plus several runs to cover corner cases might be required to achieve full coverage. In these cases, multiple log files can be provided to the analysis tool and coverage will be calculated over all of them.

The following steps need to be performed in order to verify the parallelized version and the sequential version of a program:

- **segmenting the program** -- dividing the program into logical areas (each purely parallel or sequential) by adding the corresponding API calls,
- **dumping data** -- adding API calls to the program to dump the input and output data for later comparison,
- **annotating the program** -- adding API calls to identify checkpoints, assertions and so on,
- **compiling and running** -- this will produce the required coverage and dump data,
- analyzing the results -- running the analysis tool with the previously obtained log files as inputs.

An example of the annotations on the initial *sequential version* of a hypothetical program (closely resembling the basic structure of the ray tracing application described in the next Section) is as follows:

```
int input[SIZE];
int output = 0;
// initialize input
...
// dump input and start parallel tracing
heap_report_data("area_1", "in", input);
heap_report_start_parallel("area_1");
// Iterate to perform some computation
for (i = 0; i < 10; i++) {
    char task_name[32];
    sprintf(task_name, "task_%i", i);

    heap_report_start_task(task_name);
    ... = input[i];
    // Do some work
    ...
    result = ...
    // Gather the result.
    output += result;
    heap_report_end_task(task_name);
}
// end of area
heap_report_end_parallel("area_1");
// dump output
heap_report_data("area_1", "out", output);
...
```

An example of the annotations on the *parallel version* of the same code is as follows:

```
int input[SIZE];
int output = 0;
// initialize input
...
// dump input and start parallel tracing
```

```
heap_report_data("area_1", "in", input);
heap_report_start_parallel("area_1");
// scatter and gather the data
for (i = 0; i < SIZE; i++) {
    FIFOin[i].put(input[i]);
}
for (i = 0; i < 10; i++) {
    output += FIFOout[i].get();
}
// end of area
heap_report_end_parallel("area_1");
// dump output
heap_report_data("area_1", "out", output);
...
// function executed by the i-th process
void process_func(int i)
{
    sprintf(task_name, "task_%i", i);
    heap_report_start_task(task_name);
    ... = FIFOin[i].get();
    // do work
    ...
    result = ...
    FIFOout[i].put(result);
    heap_report_end_task(task_name);
}
```

This example illustrates all previously mentioned steps. The execution of both versions of the code generates both unordered checkpoints (starting and ending of tasks) and data tracing points, which help identifying possible incorrect parallelization results, due to human or tool errors.

6.   An example: parallelizing a ray tracing application

In the following we will present the use of the HEAP toolset for the parallelization of a real life ray tracing application. Ray tracing algorithms mimic the visual process by simulating light rays from light sources, to objects, to the eye, and are a widely known as "embarrassing parallel" applications.

However, parallelizing an existing sequential implementation *without any prior knowledge* of the software and guided only by a classical source code profiler can be a daunting task. In the following we will present the use of the toolset for both top-down and bottom-up searches of parallelization opportunities and experimental results.

6.1.   Exploration for "doall" parallelization opportunities

The Graphical User Interface (GUI) of the HEAP toolset always starts by presenting a single Data Dependency Graph (DDG) node that corresponds to the main function and folds all program data flow within it (see Figure 6). The detailed information about this node presented in the pop-up window includes the percentage of estimated execution time spent in this function and its callees, its position in the call stack, and its source code reference. To help the developer in program exploration, the GUI also provides commands to directly visualize the source code corresponding to a DDG node in a pop-up or to highlight the source line in the editor of the companion IDE.
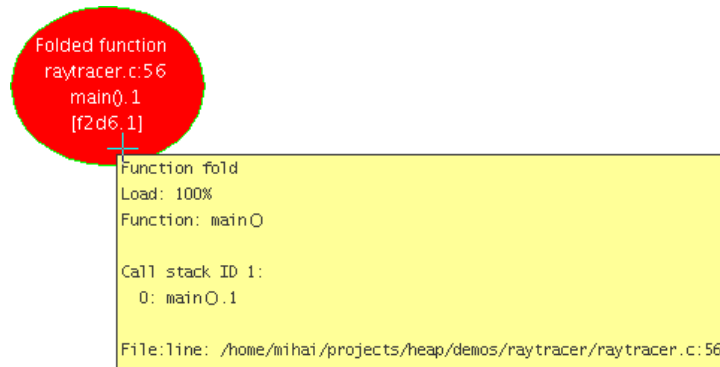
Figure 6. Initial view of the HEAP GUI displaying all program call stacks and data flows folded under the `main` *function.*

The search for parallelization opportunities starts from this top level view with:

- a progressive expansion of the nodes whose highlighted colors indicate that they include the large fractions of the program execution time, and
- an analysis of the most important data flows that connect them.

The color intensity of both nodes (representing computations) and edges (representing the data transfers between them) offers a coarse but very efficient guide towards the parts of the program that may benefit most from parallelization. The GUI can also present accurate information about the elements of interest in pop-up windows.

Figure 7 shows the unfolding of the `main` function, where it is immediately apparent that most of the program execution time is spent in the `Render` function and its callees. However, many other nodes appear in this unfold that carry too little execution weight to be interesting for the search for parallelization opportunities (such as functions for data initialization or for checking the command line switches). To prevent graph cluttering, these elements can be excluded from this and all subsequent unfolds using a special graph re-root function of the GUI. Graph re-rooting sets the selected node, usually a computational bottleneck on which the developer wants to focus, as the new root, exactly as was `main` in Figure 6.
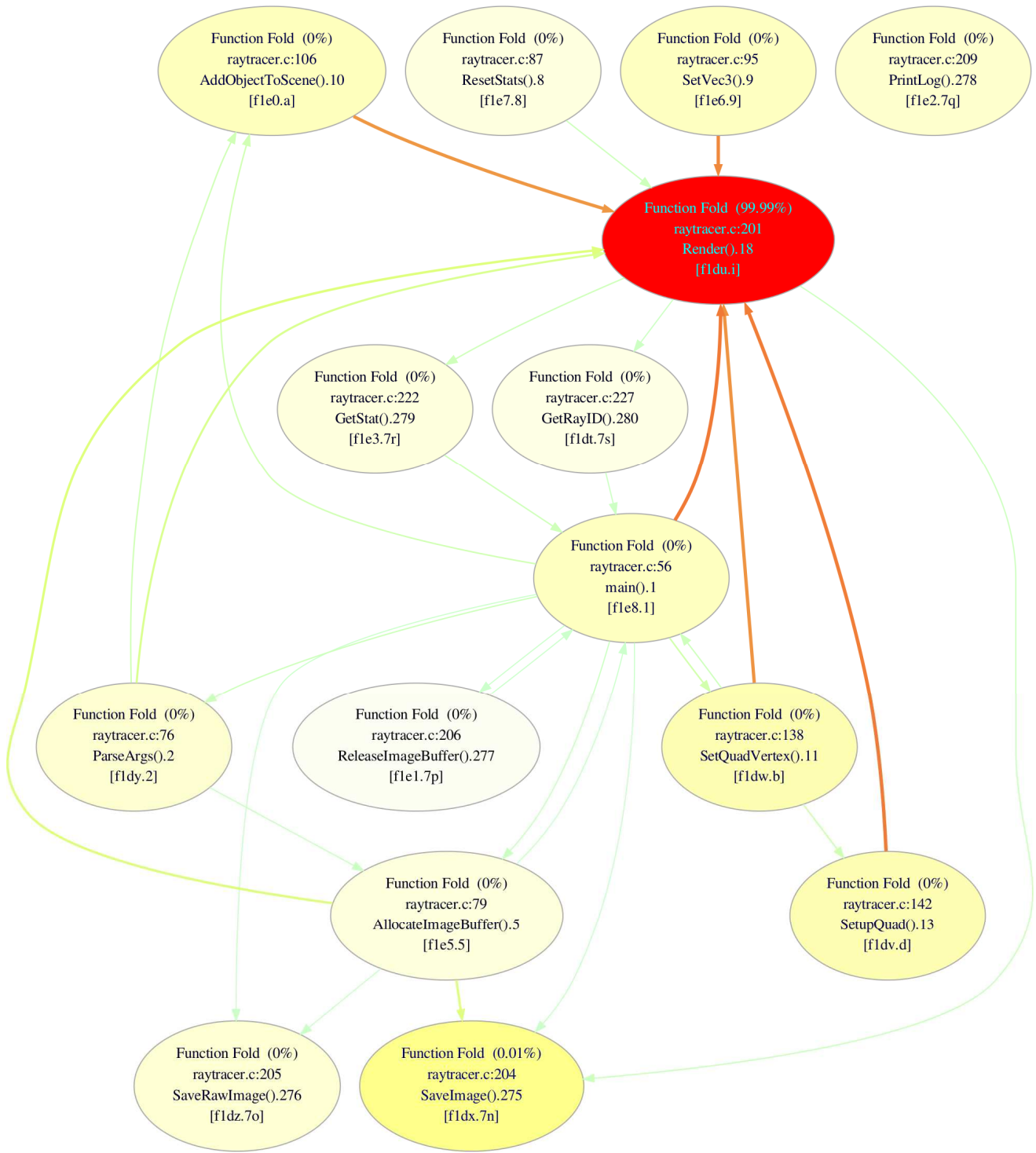
Figure 7. First level expansion of the `main` *call stack.*

The graph re-rooted on the `Render` node and subsequently unfolded is shown in Figure 8. Re-rooting restricts the display to the statements within the `Render` function, or those called by it.
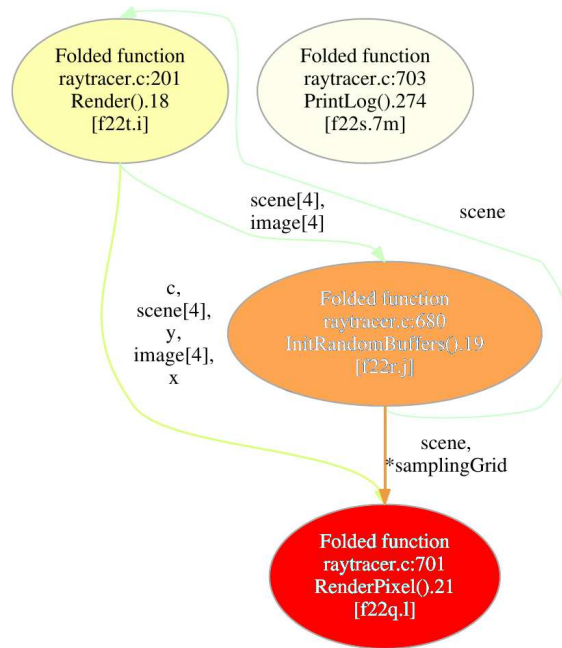
Figure 8. `RenderPixel` as a doall parallelization candidate.

Of particular interest are the calls originating in `RenderPixel` that account for most of the program execution (99.78%). The function is called within a nest of 3 loops, as shown in Figure 9, and the data dependency edges show limited dependencies outside the `RenderPixel` node.

```
for (c = 0; c < numClusters; c++)
    for(y = 0; y < blockSize; y++)
        for(x = 0; x < blockSize; x++)
            RenderPixel(scene, x, y, image, c);
```

Figure 9. Worker call in `Render` function.

Another functionality of the GUI, namely displaying only the immediate data dependencies of a function or loop body, helps to analyze in detail the data dependencies of a node which has been chosen as a parallelization candidate, and to rewrite the code in parallel form (e.g. by adding OpenMP annotations, as discussed below). Figure 10 shows the dependencies of `RenderPixel` which are either data produced outside the function and consumed inside it (inbound data dependencies) or data produced by the function and consumed outside it (outbound data dependencies).

Figure 10. Detail of the data dependency view for `RenderPixel`

To facilitate the exploration, the dependency view is organized in 5 layers as follows:

- the top layer displays the statement nodes (grouped by functions) that produce the inbound data dependencies;
- the following layer presents the inbound data dependencies as parallelogram-shaped boxes, cross-linked with their source code declarations;
- the middle layer contains all the statement nodes of the function that consume or produce the function data dependencies;
- the next layer displays the outbound data dependencies;
- the bottom layer displays the statement nodes that consume the outbound data dependencies.

Using this view, we identified the input and output dependencies of `RenderPixel` that we used to write a parallel version of the ray tracer application, using the "doall" paradigm to implement the loops mentioned above, that is presented in Section 6.3.

6.2. Exploration for "pipeline" parallelization opportunities

It is also possible to use the same GUI capabilities to explore another parallelization paradigm, namely the creation of a pipeline, in the form of a Directed Acyclic Task Graph. We can consider, for example, the graph fragment shown in Figure 11, which shows the data dependencies at a deeper level in the call stack with respect to those considered above. The DDG here clearly shows a unidirectional data flow through some functions, namely `intersectObject`, `intersectQuad` and `intersectSphere`. This provides the developer with the necessary hints about where to focus the parallelization efforts, i.e., on loops involving these functions. These unidirectional data dependencies clearly identify the *stateless parallelizable computations*.

Figure 11. Data dependency traces around the intersectObject *function of the ray tracer.*

Specifically, a quick source code inspection shows that `intersectObject` is called in exactly 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 a 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, there is no guarantee that they are only used as inputs, since both C and C++ notoriously allow to cast away `const`-ness and subsequently *update* the data structures. Regardless, the HEAP toolset data profiler allows the precise identification (within the limitation of the execution paths driven by the provided input data, of course) of which pointers are accessed as inputs and outputs. In this specific case, a more detailed inspection of the profiling data can be performed using the tools provided by the HEAP toolset graphical user interface to ascertain that the inputs are indeed only read and the output is only written.

Assuming a FIFO-based KPN structure for parallelization and assuming a goal of N-way parallelization, to match the parallelism of an N-way core, the code above can be changed 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.

It is worth noting about this parallelization that:

- can be obtained very quickly. *The entire process, including the debugging, took less than two hours for a programmer with no previous knowledge of the ray tracing application.*
- 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.

The Compaan tool 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 presented above is for a loop performed over objects.

6.3. Experimental results

Section 6.1 discovered a suitable parallelization opportunity in the call of the `RenderPixel` function and a detail analysis of its immediate data dependencies. With this information we were able to write a parallel version of the ray tracer application using two methods:

1. OpenMP annotations (as implemented by the GCC compiler), and
2. POSIX threads (pthreads) with manual synchronization.

The code in Figure 9 is parallelized for OpenMP as follows:

```
#pragma omp for private(c, x, y) schedule(static)
    for (c = 0; c < numClusters; c++)
        for(y = 0; y < blockSize; y++)
            for(x = 0; x < blockSize; x++)
                RenderPixel(scene, x, y, image, c);
```

where all data dependencies not declared `private` are considered `shared` and a static schedule means that all iterations of the outermost loop are divided between the worker threads at compile time and each worker thread will execute only its predefined segment of iterations. A dynamic schedule assigns to each worker thread a new loop iteration as soon as the thread is available to perform more work. This approach, just like the KPN approach used by Compaan, can be used with loops whose bounds cannot be defined at compile time.

The pthread version was implemented using a similar approach, but using explicit synchronization instead of OpenMP pragmas. A worker function was defined as follows:

```
void *RenderPixel_wrapper(void *thr_data)
{
  struct thr_data_s *td = (struct thr_data_s *)thr_data;
  uint32_t c, x, y;

  for (c = td->c_start; c < td->c_stop; c++)
    for (y = 0; y < td->blockSize; y++)
      for (x = 0; x < td->blockSize; x++)
        RenderPixel(td->scene,x,y,td->image,c);

  return NULL;
}
```

and the workers were instantiated as threads that execute this function, receiving the appropriate parameters using the `thr_data` structure.

Figure 12 compares the performance of a first parallelization attempt on a desktop PC with 4 cores (with an Amdahl's law-derived maximum speedup close to 4X).
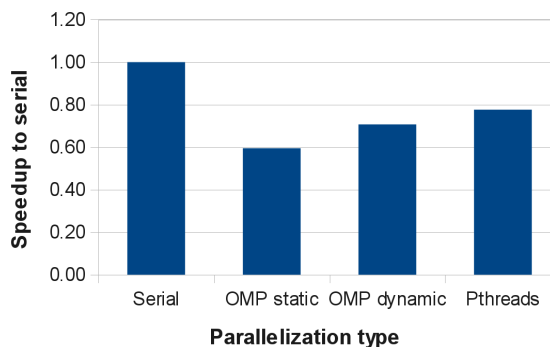

Figure 12. Execution time for different parallel versions of the ray tracer program compared with the original serial version.

As can be seen, all parallelized versions run slower than the serial version. One of the reasons for the slowdown was identified to be the false sharing occurring on some shared variables of the ray tracer program.

False sharing is an architectural effect that can occur in shared memory multiprocessor systems with coherent caches. As the data status in the caches is maintained at the block (or line) level, a write by any of the threads to a variable in a block will invalidate all copies of the block in the other caches, thus forcing a synchronization through the main memory. All threads accessing even unrelated variables in the block are blocked for the duration of the synchronization leading to a significant slowdown.

In the ray tracing application we found that a shared array of 4 global counters may lead to this problem. To prevent false sharing, we ensured that each counter is located in a different share block, by spacing them apart by at the size of a cache block of the target architecture. The speedup results are shown in Figure 13, and now all parallelized versions run faster than the serial version. Further investigations of these effects can then be performed using more traditional tools that have not been developed specifically in the HEAP project, such as Intel VTune or OProfile.
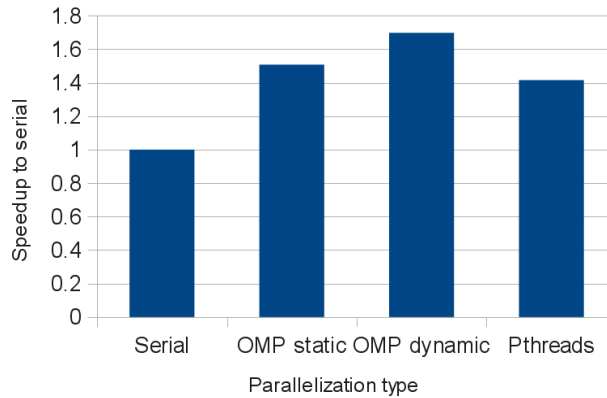

Figure 13. Execution time of the ray tracer program optimized for parallel execution.

Note that the parallelization attempts using coarse-grain *static* OpenMP and Pthreads schedules achieve less speedup than those using *dynamic* scheduling, although the latter has more run-time overhead. This is because the workload of each iteration of the parallelized loop is not balanced, as shown in Figure 14.
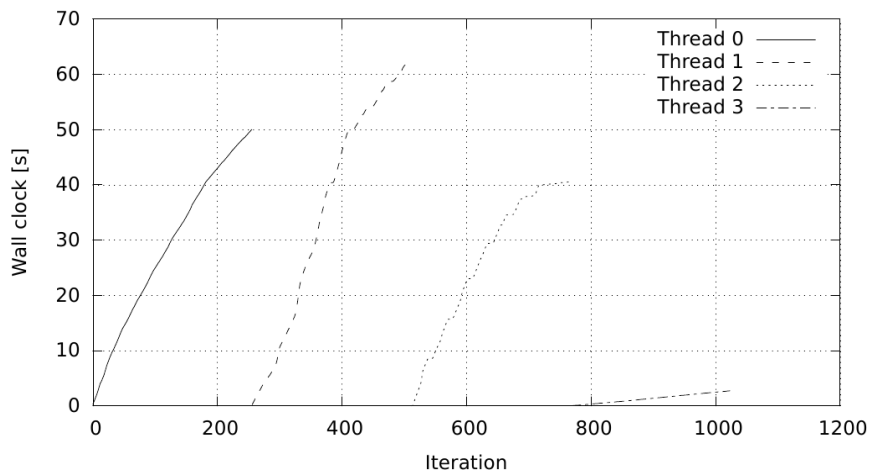

Figure 14. Time per iteration for ray tracer application parallelized using OpenMP static scheduling.

Thread 1 has a workload about 20 times larger than thread 3, because it completes its 256 loop iterations in 62 seconds, versus about 3 seconds for thread 3. Dynamic scheduling avoids this underutilization of resources by assigning a new task to threads as soon as they become idle. The OpenMP static schedule reaches a similar level of efficiency if the scheduler granularity is reduced from 256 to 10 iterations per thread (while dynamic scheduling associates each thread with 1 iteration), for instance.

7. Conclusions and future work

This ray tracing application case study shows how the HEAP approach can be used to discover multiple parallelization opportunities, leaving to the developer the choice of which suits best the underlying multi-core architecture.

This paper described a flexible multi-paradigm approach to the very difficult task of software parallelization. We discussed how potential parallelism can be identified starting both from a formal automated analysis of array indices within loops and from a data dependency execution profile. We explained how both the code changes required to apply the first approach and the manual parallelization changes required by the second approach can be verified by using a metric-driven approach.

Finally, we illustrated with a simple but realistic example, a ray tracing application, how different parallelization options can be obtained and quickly explored with the HEAP approach. Several parallelization techniques were implemented and their performance was compared with that of the serial version. The inefficiencies were analyzed and correlated with the application structure and the complexity of the task.

It is also worth noting 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, but the HEAP profiler may 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.

Future work will include (1) more extensive experimentation, using other real life applications with the analysis of the actual speedup obtained by parallelization (2) better support for the designer in the analysis of the trace data, and (3) support for automatic rewrite of the code for parallelization.

8. Acknowledgments

References

[1] Compaan Design. BV, 2012. See http://www.compaandesign.com/.

[2] B. Kienhuis, E. Rijpkema, and E. F. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in Proceedings of the Eighth International Workshop on Hardware/Software Codesign, pp. 13–17, 2000.

[3] W. Thies, V. Chandrasekhar, and S. P. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 356–369, 2007. http://www.altera.com/corporate/news_room/releases/2011/products/nr‒opencl.html?GSA_pos=5&WT.oss_r=1&WT.oss=OpenCL

[4] J.-Y. Mignolet, R. Baert, T. J. Ashby, P. Avasare, H.-O. Jang, and J. C. Son, "Mpa: Parallelizing an application onto a multicore platform made easy," IEEE Micro, vol. 29, no. 3, pp. 31–39, 2009.

[5] S. Kaxiras and G. Keramidas, "Sarc coherence: Scaling directory cache coherence in performance and power," IEEE Micro, vol. 30, no. 5, pp. 54–65, 2010.

[6] G. Kahn, "The semantics of a simple language for parallel programming," in Proceedings of IFIP Congress, Aug. 1974.

[7] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," ACM Comput. Surv., vol. 26, pp. 345–420, Dec. 1994.

[8] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M.Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: an infrastructure for research on parallelizing and optimizing compilers," SIGPLAN Not., vol. 29, pp. 31–37, Dec. 1994.

[9] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," ACM Comput. Surv., vol. 27, pp. 367–432, Sept. 1995.

[10] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in split-c," in Supercomputing '93. Proceedings, pp. 262 – 273, nov. 1993.

[11] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. Cronquist, and M. Sivaraman, "Pico: automatically designing custom computers," Computer, vol. 35, pp. 39 – 47, sep 2002.

[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on, pp. 356 –369, dec. 2007.

[13] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in Int'l Conference on Compiler Construction, pp. 213–228, 2002.

[14] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4, Sedms'93,