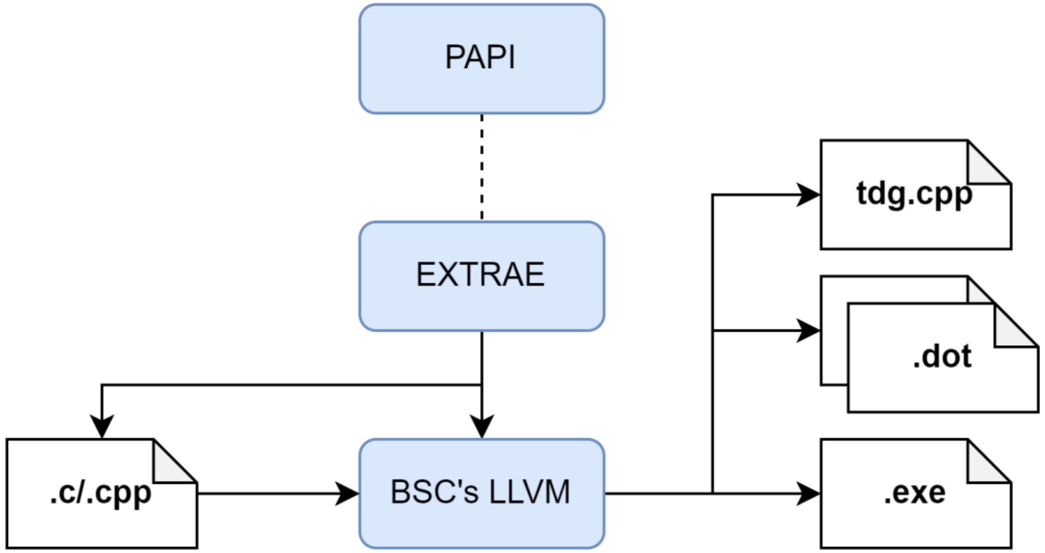
1. Compilation Phase

- **Requires:** PAPI, Extrae and BSC's LLVM
- **Input**: c/cpp + OpenMP code
  - Instrument with extrae calls and taskgraph generation pragmas
- **Output**: executable and TDG-related files



PAPI

- Performance Counters extraction API
  - Counter inside Hardware/Software Components (CPU, GPU, accelerators,…)
  - Device-dependent

- Allows us to see selected HW/SW counters around a specific code parcel
  - E.g. evaluate performance of OpenMP tasks!

- To reduce intrusion, we evaluate some perf. counters:
  - Number of CYCLES
  - L1, L2 and L3 data cache access ratios (hits vs misses)

Extrae

- Monitor and extract runtime performance traces of applications

- Designed for applications compiled and run with the shared memory model (e.g. OpenMP)

- Outputs trace files in “paraver” trace format (CSV-like)

Testing Extrae

- Let's build a small example to test extrae
- Add Extrae_init and Extrae_fini calls in the first_test project

```
cd $COURSE_PATH
cd workspace/first_test
code . #add extrae calls
make test
./test
```

main.c

```
#include <stdio.h>
#include <extrae.h>
int main()
{
  Extrae_init();
  #pragma omp parallel
  #pragma omp single
  {
      #pragma omp task
      printf("Hi from task 1\n");
      #pragma omp task
      printf("Hi from task 2\n");
  }
  Extrae_fini();
}
```

Makefile

```
test: main.c
   ${CC} -I${TAFLOW_PATH}/extrae/include -L${TAFLOW_PATH}/extrae/lib main.c -fopenmp -lomptrace -o test
```

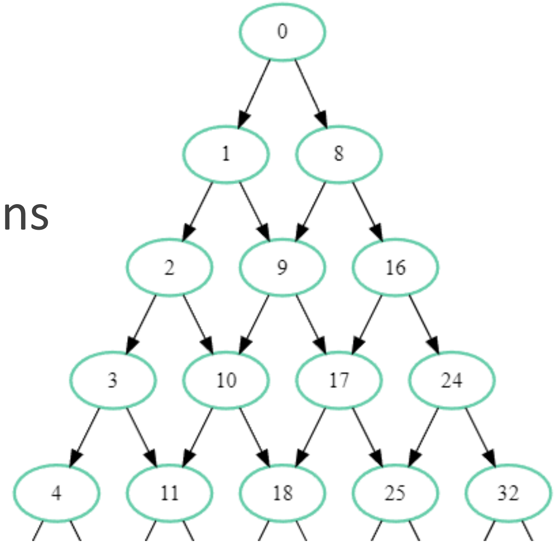
- You can also try the “heat” example provided in the exercises folder

Testing Extrae: expected output

```
Welcome to Extrae 3.8.3     #Extrae Logging
Extrae: Detected GOMP version is 4.5
Extrae: Detected and hooked OpenMP runtime: [Intel KMPC] [GNU GOMP]
Extrae: OMP_NUM_THREADS set to 4
Extrae: Parsing the configuration file (/opt/taflow/extrae/share/example/OMP/extrae.xml)
.... #if errors with counters, just ignore them for now!
Extrae: Error! Hardware counter PAPI_TOT_INS (0x80000032) is not available. Check set 1.
… #more logging
Extrae: Tracing mode is set to: Detail.
Extrae: Successfully initiated with 1 tasks and 4 threads
                #
Hi from task 1  # application output
Hi from task 2  #
Extrae: Intermediate raw trace file created : .../test/set-0/TRACE@mypc.0000087494000000000000.mpit
...                            #more Extrae Logging
mpi2prv: Congratulations! test.prv has been generated.
```

BSC's LLVM

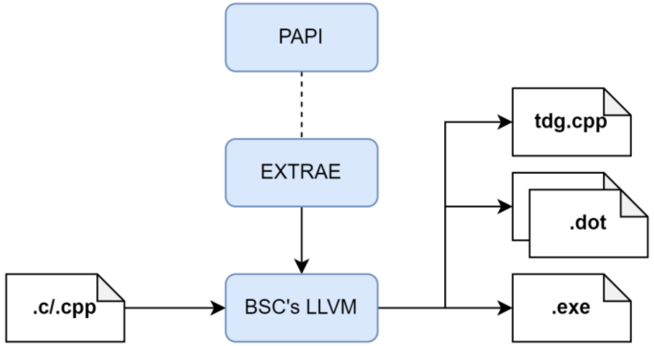
- C/CPP compilation with extra features in OpenMP
- Features designed for task-based OpenMP applications with dependencies
- Generates Task-Dependency Graphs (TDGs) from the OpenMP parallel regions
  - Simplifies the parallel DAG problem
  - All tasks are "siblings"
  - Connections represent dependencies
- Allows static task-to-thread mapping
- TDG is generated in
  - .dot format: ideal for analysis
  - .cpp format: ideal to apply static mapping



Example of a generated TDG

LLVM Compilation

- We can use the previous example to build the same application
- But first, let’s make some changes



main.c

```
…

int main()
{
  Extrae_init();
  #pragma omp parallel
  #pragma omp single
  #pragma omp taskgraph tdg_type(static)
  {
     …
  }
  Extrae_fini();
}
```

Makefile

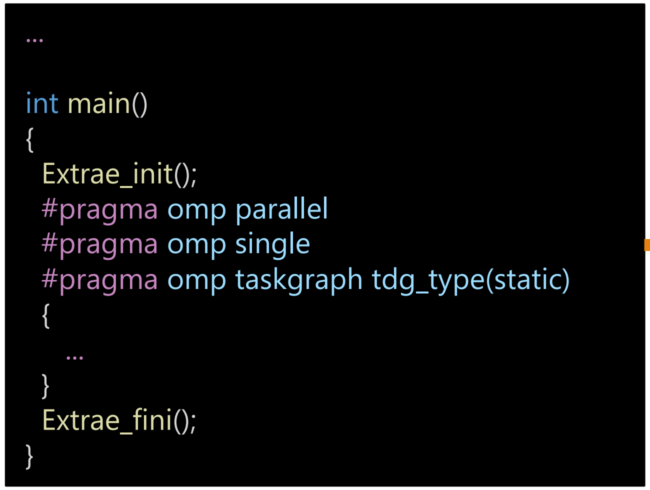
```
TDG = -fopenmp-taskgraph -static-tdg -L${OMP_PATH}

test: main.c
     ${CC} … main.c ${TDG} -fopenmp -lomptrace -o test
```

- Hints the compiler to extract a TDG

LLVM Compilation

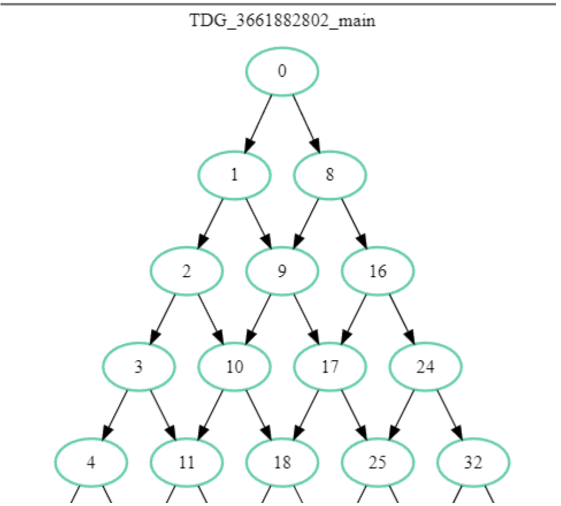
- The compilation now generates extra files:
  - *_tdg_*.dot: TDG structure as a Digraph
  - *_tdg.cpp: TDG as a C-style structure
  - tdg.hpp: contains TDG struct

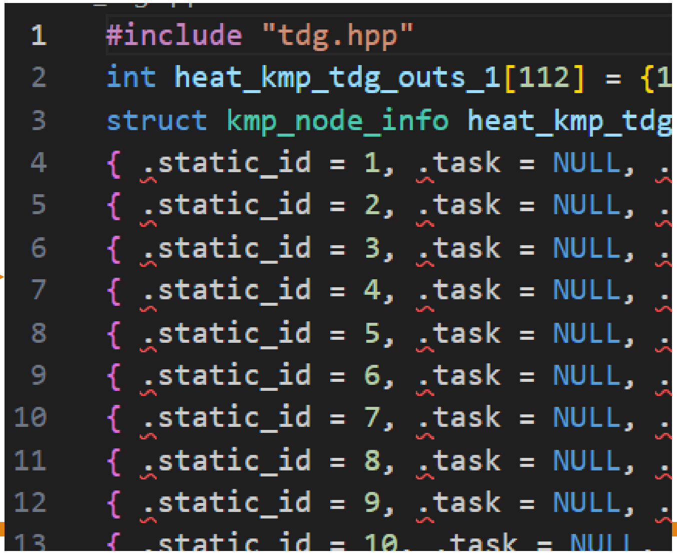
- Check if these files were generated



```
…

int main()
{
  Extrae_init();
  #pragma omp parallel
  #pragma omp single
  #pragma omp taskgraph tdg_type(static)
  {
     …
  }
  Extrae_fini();
}
```

LLVM

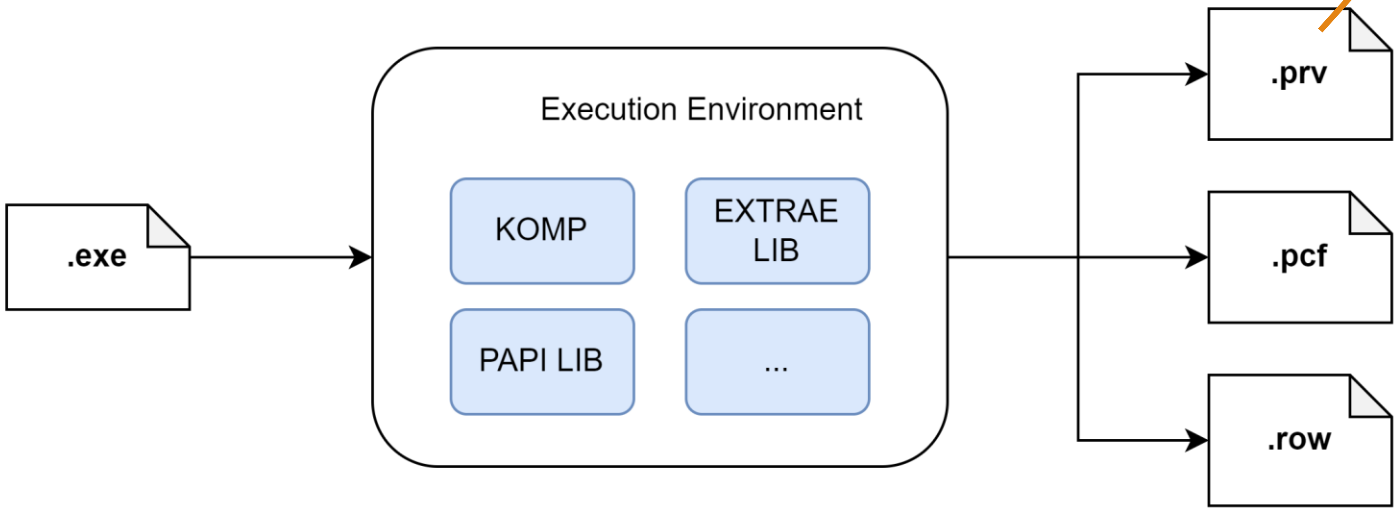




```
#include "tdg.hpp"
int heat_kmp_tdg_outs_1[112] = {1
struct kmp_node_info heat_kmp_tdg
{ .static_id = 1, .task = NULL, .
{ .static_id = 2, .task = NULL, .
{ .static_id = 3, .task = NULL, .
{ .static_id = 4, .task = NULL, .
{ .static_id = 5, .task = NULL, .
{ .static_id = 6, .task = NULL, .
{ .static_id = 7, .task = NULL, .
{ .static_id = 8, .task = NULL, .
{ .static_id = 9, .task = NULL, .
{ .static_id = 10, .task = NULL,
```

Running the Application

- Execution (and output) with the new LLVM is still the same



- Open this file and analyse it

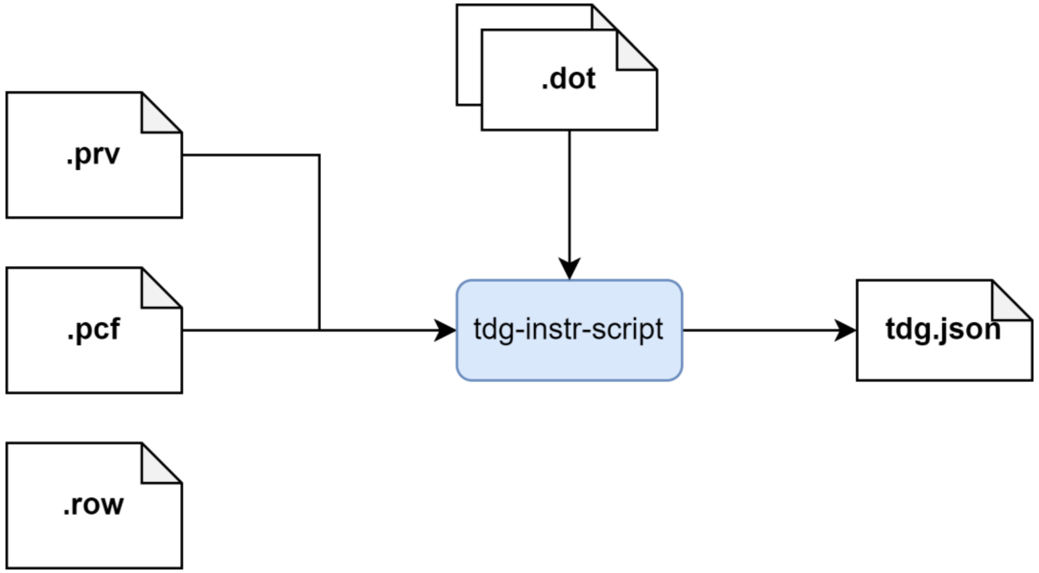
- The output trace file (prv)
  - Mainly designed for GUI-based BSC Tools (paraver, which roughly translates to “to see”)
  - Hard to read/understand (trace e.g. 2:1:1:1:1:0:40000018:1:41999999:1:42000050:0…)
  - Difficult to process/automate analysis
  - One file per execution (rewrites file)

Converting Extrae Output

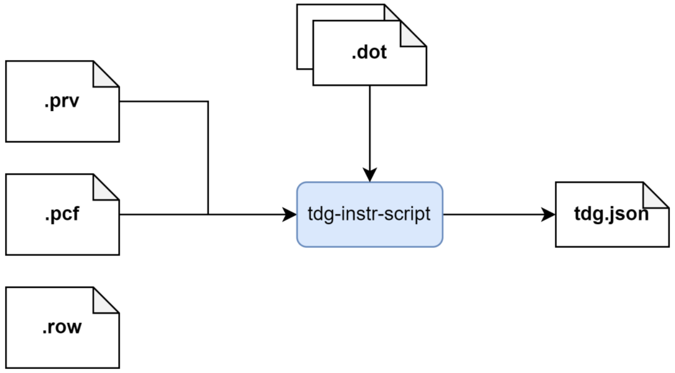
- Using the python script: **tdg-instrumentation-script**

- Converts prv file into json, which contains:
  - parallel regions containing tasks (“nodes”)
  - nodes containing dependencies (“ins”,”outs”) and results

- More readable and easier to parse/manipulate



Converting Extrae Output



- Conversion (first execution): generates tdg_output.json file

```
python3 ${TAFLOW_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py *.dot *.prv *.pcf
```

- Update JSON file (following executions)

```
python3 ${TAFLOW_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py tdg.json *.prv *.pcf
```

TDG.json Example

```
{
    "test": [  #application name
        {
            "taskgraph_id": 2658744759, #uid for the TDG
            "nodes": {
                "0": { #one “node” for each OpenMP Task
                    "ins": [],  "outs": [], #in and out dependencies of the Task
                    "results": [
                        {  #one result per execution of the OpenMP Task
                            "thread": 1,
                            "execution_begin_time": 18996782,
                            "execution_end_time": 19108462,
                            "execution_total_time": 111680,
                            "42000050": 5196,   #a performance counter (see .pcf for the name)
```

2. Profiling Phase

- A series of executions of the application

- Each execution providing a .prv file with 1 or more results
  - One for each execution of the parallel region
  - Loop in the application can be used for multiple results in same execution

- Prv file must be converted into the JSON file
  - Or results appended to an existing one

- Final result will be a JSON file
  - organized by tasks
  - each task with an array of results

2. Profiling Phase Example

- Run the following commands
  - Observe the evolution of the tdg_output.json file between each command

```
cd $COURSE_PATH
cd workspace/first_test
make test
./test
python3 ${TAFLOW_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py *.dot *.prv *.pcf
./test
python3 ${TAFLOW_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py tdg_output.json *.prv *.pcf
./test
python3 ${TAFLOW_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py tdg_output.json *.prv *.pcf
…
```

- Ideally, it is better to automate this process (e.g. via a script)