Lab 1: Experimental setup and tools

par4111
Adrià Cabeza, Xavier Lacasa
Departament d' Arquitectura de Computadors

February 26, 2019
2018 - 19 PRIMAVERA

Contents

1 Introduction

In order to do properly this subject, first, we have to introduce some new concepts and hardware and software environment that we will use during this semester to do all laboratory assignments. The following document contains an introductory approach, step by step introducing those concepts. We will introduce the *Boada* architecture, some of the most important parallelism concepts and several tests to see its effects.

2 Experimental setup

2.1 Node architecture and memory

Boada is a multiprocessor server located at the Computer Architecture Department divided in different nodes, each of them with different architecture and diffferent uses. *Boada* is composed of 8 nodes (from boada-1 to boada-8) and they can be grouped as the following table:

Node name	Processor generation	Interactive	Queue name
boada-1	Intel Xeon E5645	Yes	batch
boada-2 to 4	Intel Xeon E5645	No	execution
boada-5	Intel Xeon E5-2620 v2 + Nvidia K40c	No	cuida
boada-6 to 8	Intel Xeon E5-2609 v4	No	execution2

However in this course we are going to use mainly from boada-1 to boada-4. The easiest way to obtain the information of the hardware used in each node is using the linux commands lscpu and lstopo(1 and 2). This commands can be easily executed in the boada-1 node (because it is interactive), but if we want to use the other nodes we can use the submit-*.sh script provided by the PAR professors and use the queue system.

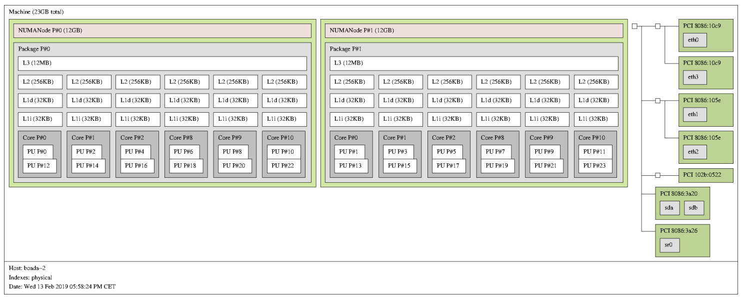

Figure 1: Boada-2 architecture outputed by lstopo.

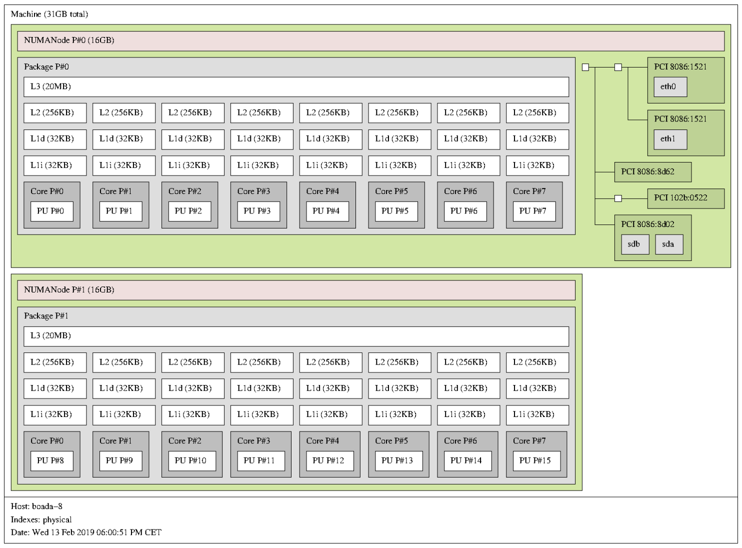

Figure 2: Boada-8 architecture outputed by lstopo.

After creating the scipts and applying them to each of the nodes, we obtained the following hardware information:

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
L1-I cache size (per-core)	32 KB	32 KB	32 KB
L1-D cache size (per core)	32 KB	32 KB	32 KB
L2 cache size (per-core)	256 KB	256 KB	256 KB
Last-level cache size (per-socket)	12 MB	15 MB	20 MB
Main memory size (per socket)	12 GB	31 GB	16 GB
Main memory size (per node)	23 GB	63 GB	31 GB

The previous table gives us useful information that will be necessary in the future to properly use the *boada* system and understand the parallelism decomposition and time we will get.

2.2 Sequential and parallel executions

More often than not parallelism offers speed-ups in the execution time of applications. Sometimes, however, that extra speed is used to augment the problem size, which would not be possible otherwise.

In the two following sections we are going to see the differences of two different approaches to parallelism, **strong** and **weak**, applied to the *pi_omp.c* program.

2.2.1 Strong scalability

Strong scalabilty consists in increasing the numberer of processors while keeping the problem size the same. This reduces the amount of work each processor has to do, which speeds-up the execution. Nonetheless, the speed-up is bounded by the parallelization of the program and the overhead generated when doing so. Usually a point is reached where adding processors has no further effect on the program or the overhead generated by further parallelizing the program is greater than the added speed-up. ARA HEM DE POSAR ELS GRÀFICS PELS DIFERENTS BOADAS PERO NO ELS PUC CREAR, ES GENEREN EN BLANC JKAHSDFUJAHFUJAHFA

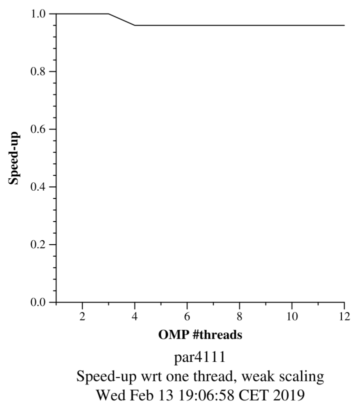

Figure 3: *pi_omp* with 100000000 weak by boada-6

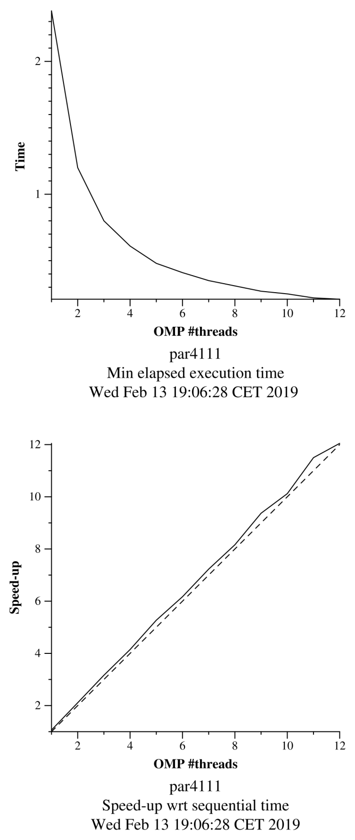

Figure 4: *pi_omp* with 1000000000 strong by boada-8

2.2.2 Weak scalability

Weak scalability takes a different approach. It takes advantage of the additional power gained by parallelizing the program to increase the problem size, so that while the speed-up stays more or less the same, the work done increases. ARA HEM DE POSAR ELS GRÀFICS PELS DIFERENTS BOADAS PERO NO ELS PUC CREAR, ES GENEREN EN BLANC JKAHSDFUJAHFUJAHF

3 Experimental setup

3.1 Introduction

The objective of this laboratory is learn how to use Tareador, an environment to analyse the potential parallelilsm that can be obtained when a certain task decomposition is applied to a code. We will introduce how it works and we will experiment and analyse decomposition with a sequential code called 3DFFT.

3.2 Analysis of task decompositions for 3DFFT

Once we have seen the basic features in *Tareador* we can now proceed to explore new tasks decompositions for a piece of code. Down below we will incrementally generate five new task decompositions and the potential parallelism (T_1/T_∞) from the task dependence graph generated by *Tareador*.
To obtain T_∞ we will assume that each instruction takes one time unit to execute and simulate the execution of the graph with a large number of processors.

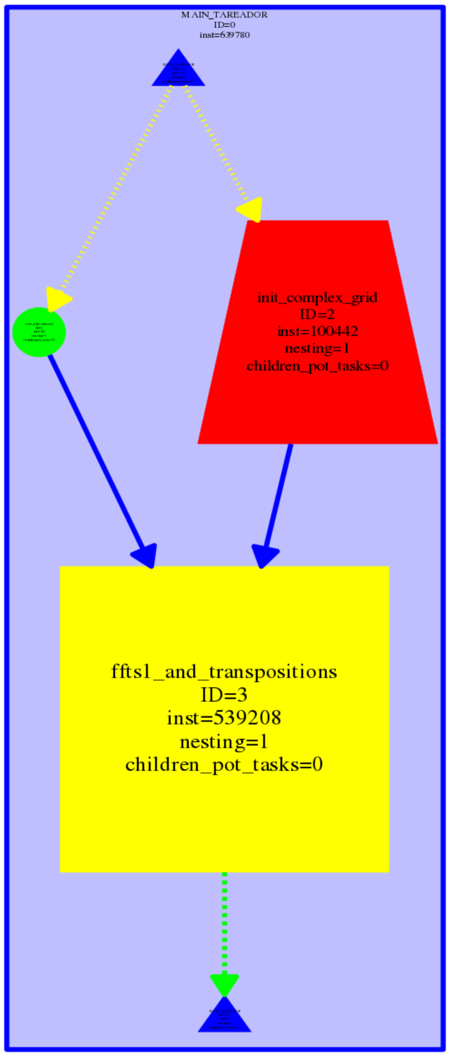

Figure 5: Dependency graph for the original version.

3.2.1 Version 1

The first version consists in replacing the task named `ffts1_and_transpositions` with a sequence of finer grained tasks, one for each function invocation inside it.

```
...
    tareador_start_task("0");
    ffts1_planes(p1d, in_fftw);
```

```
tareador_end_task("0");

 tareador_start_task ("1");
transpose_xy_planes(tmp_fftw,  in_fftw );
tareador_end_task("1");

 tareador_start_task ("2");
 ffts1_planes (p1d, tmp_fftw);
tareador_end_task("2");

 tareador_start_task ("3");
transpose_zx_planes( in_fftw ,  tmp_fftw);
tareador_end_task("3");

 tareador_start_task ("4");
 ffts1_planes (p1d,  in_fftw );
tareador_end_task("4");

 tareador_start_task ("5");
transpose_zx_planes(tmp_fftw,  in_fftw );
tareador_end_task("5");

 tareador_start_task ("6");
transpose_xy_planes( in_fftw ,  tmp_fftw);
tareador_end_task("6");
```

Once we have created all these tasks, we ... PURO PALO, ja ho faré

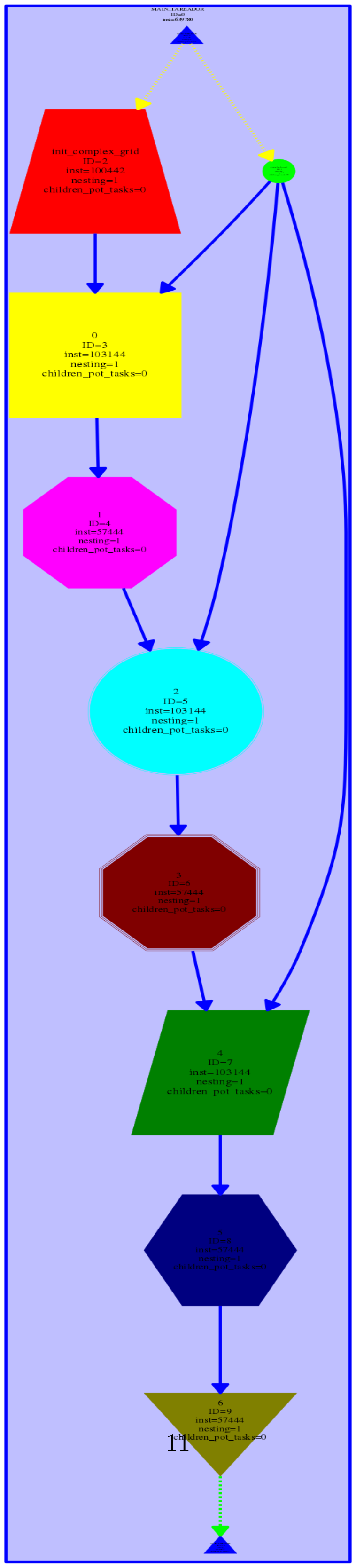

Figure 6: Dependency graph for the first version.

3.2.2 Version 2

The second version, starting from the first one, consists in replacing the definition of tasks associated to function invocations `ffts1_planes` with fine-grained tasks defined inside the function body and associated to individual iterations of the k loop.

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
     tareador_start_task("ffts1_planes_loop_k");
     for (j=0; j<N; j++) {
        fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (
            fftwf_complex *)in_fftw[k][j][0]) ;
     }
     tareador_end_task("ffts1_planes_loop_k");
    }
}


int main(){
 ...
    tareador_start_task("1");
    transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("1");

    ffts1_planes(p1d, tmp_fftw);

    tareador_start_task("3");
    transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("3");

    ffts1_planes(p1d, in_fftw);

    tareador_start_task("5");
    transpose_zx_planes(tmp_fftw, in_fftw);
    tareador_end_task("5");

    tareador_start_task("6");
    transpose_xy_planes(in_fftw, tmp_fftw);
    tareador_end_task("6");
 ...
}
```

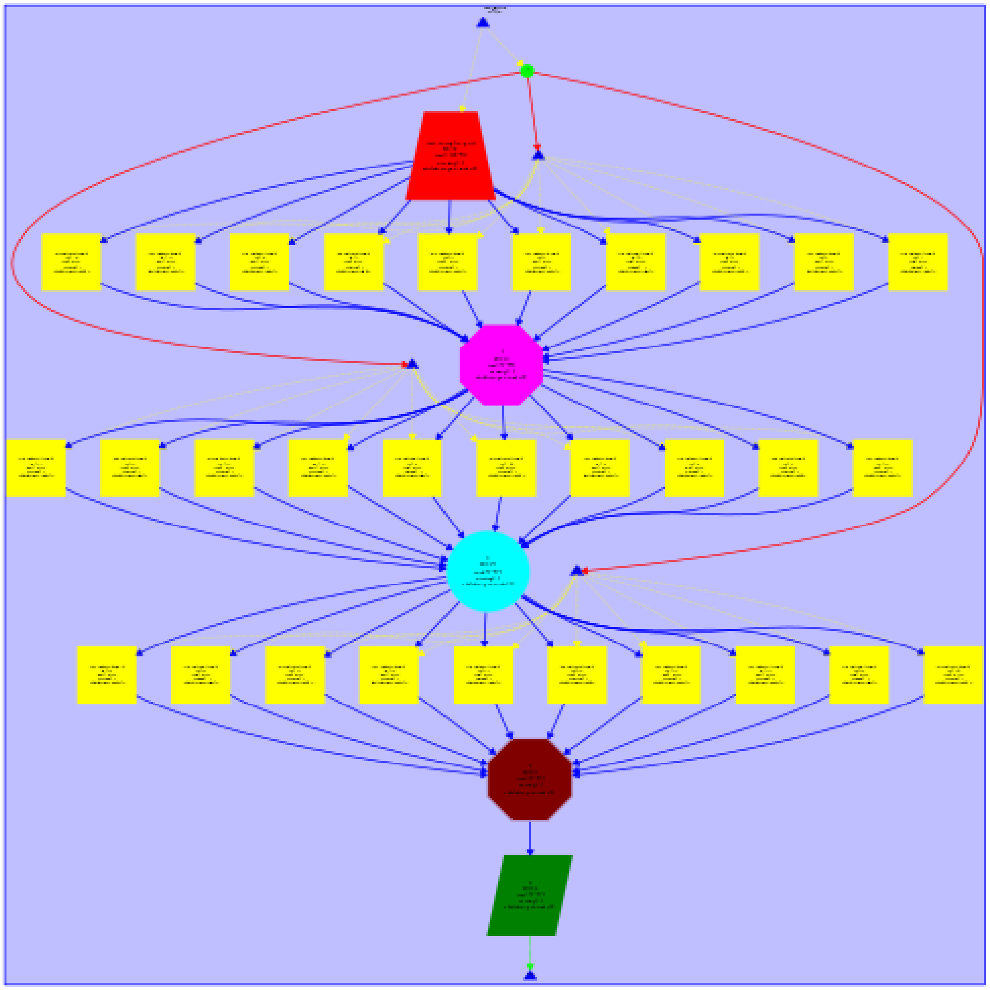

Figure 7: Dependency graph for the second version.

3.2.3 Version 3

The third version, starting from the second one, consists in replacing the definition of tasks associated to function invocations `transpose_xy_planes` and `transpose_zx_planes` with fine-grained tasks inside the corresponding body functions and associated to individual iterations of the k loop, similarly it was made in the second version.

```
void transpose_xy_planes(fftwf_complex tmp_fftw[][N][N], fftwf_complex
     in_fftw [][N][N]) {
    int k,j,i;

    for (k=0; k<N; k++) {
      tareador_start_task ("transpose_xy_planes_loop_k");
      for (j=0; j<N; j++) {
        for (i=0; i<N; i++)
        {
          tmp_fftw[k][i][j][0]  = in_fftw[k][j][i][0];
          tmp_fftw[k][i][j][1]  = in_fftw[k][j][i][1];
        }
      }
      tareador_end_task("transpose_xy_planes_loop_k");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex
      tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
     tareador_start_task ("transpose_zx_planes_loop_k");
    for (j=0; j<N; j++) {
      for (i=0; i<N; i++)
        {
          in_fftw[i][j][k][0]  = tmp_fftw[k][j][i][0];
          in_fftw[i][j][k][1]  = tmp_fftw[k][j][i][1];
        }
     }
     tareador_end_task("transpose_zx_planes_loop_k");

    }
}

int main(){
 ...
     tareador_start_task ("init_complex_grid");
    init_complex_grid(in_fftw);
    tareador_end_task("init_complex_grid");

    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

     ffts1_planes (p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
     ffts1_planes (p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
     ffts1_planes (p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);
 ...
}
```

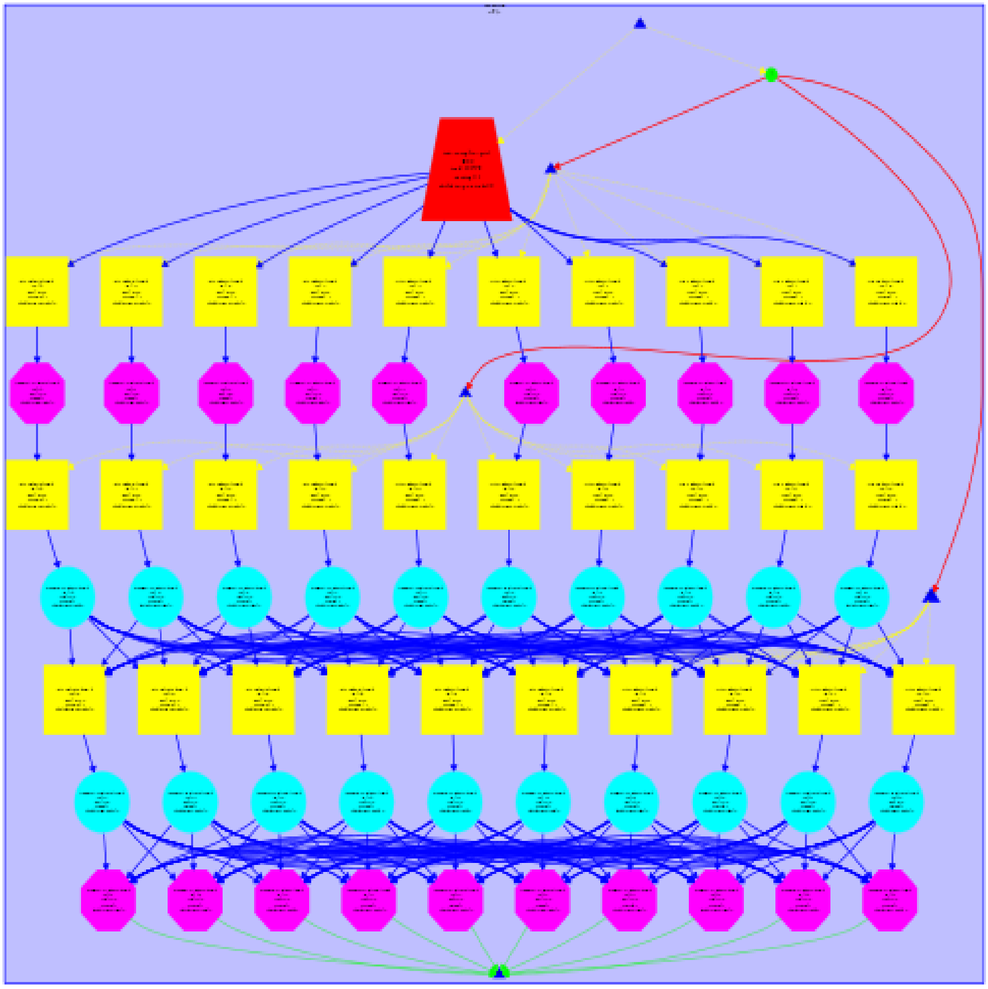

Figure 8: Dependency graph for the third version.

3.2.4 Version 4

```
void init_complex_grid(fftwf_complex in_fftw [][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    tareador_start_task("transpose_init_complex_grid_loop_k");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++)
      {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI
            *((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
    }

     tareador_end_task("transpose_init_complex_grid_loop_k");
 }
}
int main(){
 ...

    init_complex_grid(in_fftw);
    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

     ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
     ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
     ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);
 ...
}
```

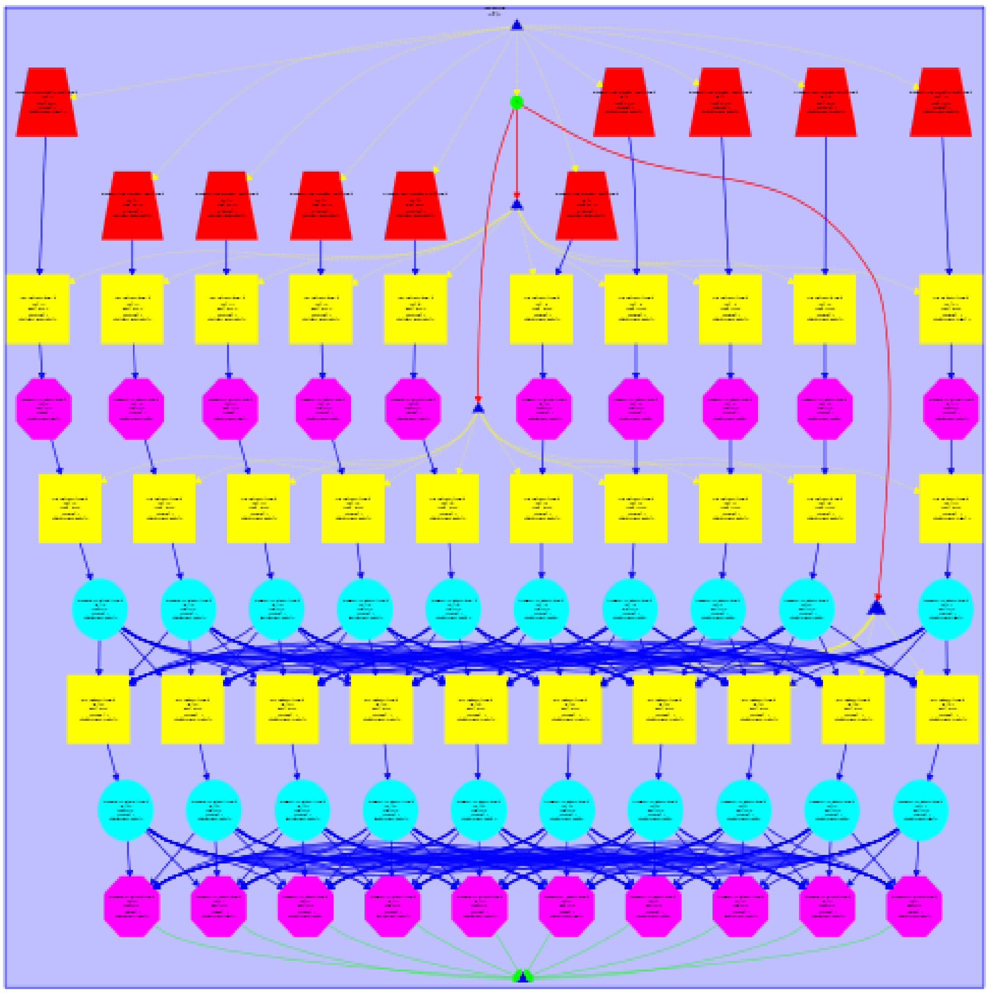

Figure 9: Dependency graph for the forth version.

3.2.5 Version 5

Version	T_1	T_∞	Parallelism
seq			
v1			
v2			
v3			
v4			
v5			

4 Conclusions