Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

Chapter 6

Experimental Studies

In this chapter, we examine the approach presented in Chapter 3 by deriving a PPN parallel specification from a real-life application called Low Speed Obstacle Detection (LSOD). This application contains relaxation II described in Section 1.2. We present some of the results we have obtained by implementing and executing the derived parallel PPN specification of the LSOD application on a shared memory multi-processor system. The main objective of this experiment is to show the practical applicability of our approach on a real-life application and to show the benefits of our parallelization approach. For the implementation, we derive the PPN specification of the LSOD application following the approach presented in Chapter 3. Then, we use the ESPAM [55] tool and the HDPC [56] back-end library to generate C++ code for the processes and the FIFO channels.

In this chapter, in Section 6.1 we evaluate our parallelization approach presented in Chapter 3, and in Section 6.2 we present the conclusions.

6.1 Low Speed Obstacle Detection

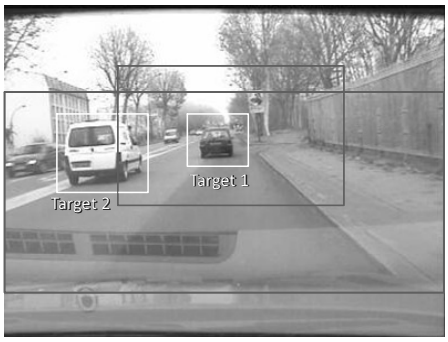
The LSOD application shown in Figure 6.1(a) is intended to detect and to track objects in front of a car in traffic. The output of the system presents spatial positions for targets – cars, pedestrians, etc. Applying several general image processing algorithms helps to find new targets, and to track existing targets. The algorithms implement shadow detection, symmetry detection, lights detection, motion segmentation, and vertical edge detection. The output of each algorithm is collected by a particle filter component [24] for analysis.

The first step in the LSOD application is to obtain two images from a given camera picture. They are named high and low resolution images and are depicted by the

two dark rectangles in Figure 6.1(b). Applying different image processing algorithms on these images, hypotheses whether cars exist are computed. Possible targets are defined as coordinates and dimensions of rectangles belonging either to the high or low resolution image. In Figure 6.1(b), two possible targets are presented by the white rectangles, surrounding the cars. Then, for every identified target, the image gradient in vertical direction of the area of the target is computed. The result is finally analyzed in order to support or decline a target.

```
0 vsum[] = 0
1 for k = 1 to Targets,
2   [Height,Width,X,Y] = getLSODTarget(k)
3   for j = 0 to Height+1,
4     for i = 0 to Width+1,
5       img[j,i] = readTarget(X,Y)
6     endfor
7   endfor
8   for j = 1 to Height,
9     for i = 1 to Width,
10      img_out[j,i] = edgeDetection(
                       img[j-1,i-1],img[j-1,i+1],
                       img[j  ,i-1],img[j  ,i+1],
                       img[j+1,i-1],img[j+1,i+1])
11      img_out[j,i] = absVal( img_out[j,i] )
12    endfor
13  endfor
14  for j = 1 to Height,
15    for i = 1 to Width,
16      vsum[i] = vertSum( vsum[i], img_out[j,i] )
17    endfor
18  endfor
19 endfor
```

(a) Pseudo code of the edge detection part of the motivating example. Target size is specified by variables `Height` and `Width`.



(b) LSOD applied on real data. The vehicles in front of the camera are detected and tracked. The dark rectangles depict the area of the image that is processed.

Figure 6.1: Pseudo code of the edge detection part of the LSOD application and its application on real data.

The edge detection part of the LSOD application, shown in Figure 6.1(a), is an example of a program which is not a static affine nested loop program. This program is affine nested loop program but it has dynamic control as function `getLSODTarget()` at line 2 initializes variables `Height` and `Width` used as loop bounds. These variables define the size of a target, i.e., the amount of data to be processed, and change values for every target at run-time. Since targets are moving in front of a camera (which is also moving), the identified positions stored in variables `(X,Y)` and dimensions `(Height,Width)` will differ for different targets in the frame and for one and the same target in different frames. That is why, the values of variables `Height` and `Width` (as well as the number of targets) are unknown at compile-time, and therefore, the *pn* compiler [3] cannot handle the program shown in Figure 6.1(a).

Parallelization

The result of Steps 1 to 3 of the solution approach presented in Chapter 3 and applied on the LSOD program in Figure 6.1(a) is illustrated in Figure 6.2. It is the final dSAC specification. We use the final dSAC to derive the PPN topology, shown in Figure 6.3, as well as to derive the internal code structure of all processes. Examples of the internal code structures of processes `readTarget` and `edgeDetection` are depicted in Figure 6.4. Below, we emphasize on some of the important moments of the PPN derivation, we describe the PPN topology, show how code for processes is generated, and comment on the overhead introduced in the generated PPN.

```
1  for k = 1 to Targets,
2    [Height,Width] = getLSODTarget()
3    X_j[k] = Height
4    X_i[k] = Width
5    for j = 0 to maxHeight + 1,
6      for i = 0 to maxWidth + 1,
7        if (j <= X_j[k] + 1 && i <= X_i[k] + 1)
8          img_1[k,j,i] = readTarget(X,Y)
9          lcl_1[j,i] = (j,i)
10       endif
11       ctrl_1[k,j,i] = lcl_1[j,i]
12     endfor
13   endfor
14   for j = 1 to maxHeight,
15     for i = 1 to maxWidth,
16       (c11,c12) = ctrl_1[k,j-1,i-1]
17       if (c11 == j-1 && c12 == i-1)
18         in_0 = img_1[k,j,i]
19       endif
20       (c21,c22) = ctrl_1[k,j-1,i+1]
21       if (c21 == j-1 && c22 == i+1)
22         in_1 = img_1[k,j,i]
23       endif
24       (c31,c32) = ctrl_1[k,j,i-1]
25       if (c31 == j && c32 == i-1)
26         in_1 = img_1[k,j,i]
27       endif
28       (c41,c42) = ctrl_1[k,j,i+1]
29       if (c41 == j && c42 == i+1)
30         in_1 = img_1[k,j,i]
31       endif
32       (c51,c52) = ctrl_1[k,j+1,i-1]
33       if (c51 == j+1 && c52 == i-1)
34         in_1 = img_1[k,j,i]
35       endif
36       (c61,c62) = ctrl_1[k,j+1,i+1]
37       if (c61 == j+1 && c62 == i+1)
38         in_5 = img_1[k,j,i]
39       endif
40       if (j <= X_j[k] && i <= X_i[k])
41         img_out_1[k,j,i] = edgeDetection(
                                 in_0, in_1,
                                 in_2, in_3,
                                 in_4, in_5)
42         img_out_2[k,j,i] = absVal( img_out_1[k,j,i] )
43         lcl_2[j,i] = (j,i)
44       endif
45       ctrl_2[k,j,i] = lcl_2[j,i]
46     endfor
47   endfor
48   for j = 1 to maxHeight,
49     for i = 1 to maxWidth,
50       (c71,c72) = ctrl_2[k,j,i]
51       if (j == c71 && i == c72)
52         in_0 = img_out_2[k,j,i]
53       endif
54       if (j <= X_j[k] && i <= X_i[k])
55         if( j >= 1 )
56           in_1 = vsum[i]
57         else
58           in_1 = 0
59         endif
60         vsum[i] = vertSum( in_1, in_0 )
61       endif
62     endfor
63   endfor
64 endfor
```

Figure 6.2: Final dSAC.

According to Step 1 (see Section 3.2) of our solution approach we substitute the dynamic upper bound functions with constants equal to the maximum values these functions can have. The initial LSOD program in Figure 6.1(a) has three loop nests with dynamic upper bound functions: `Height+1`, `Width+1`, `Height` and `Width` at lines 3–4, 8–9 and 14–15, respectively. These functions take their maximum when variables `Height` and `Width` are maximum, i.e., equal to some constants `maxHeight` and `maxWidth`, respectively. In the LSOD program, the maximum values of `Height` and `Width` are the maximum dimensions that a target may have. Therefore, we substitute

the dynamic upper bound functions with constants equal to the maximum of these functions as depicted in Figure 6.2 at lines 5–6, 14–15 and 48–49.

The result of applying the FADA analysis which constitutes Step 2 (see Section 3.3) of our solution approach is illustrated at lines 17–19, 21–23, 25–27, 29–31, 33-35, 37–39 and 51-53 in Figure 6.2. In total, 6 two-dimensional vectors of parameters $(c11, c12), \ldots, (c61, c62)$ were introduced by the FADA algorithm analyzing the data-dependencies between functions `readTarget()` and `edgeDetection()` shown in Figure 6.1(a). Also, one two-dimensional vector of parameters $(c71, c72)$ was introduced after analyzing the data-dependencies between functions `absVal()` and `vertSum()`.

At Step 3 (see Section 3.4) of our solution approach, we introduce local and global control arrays in order to initialize and propagate the values of the parameters at run-time. For the pair of functions `readTarget()` and `edgeDetection()`, 6 vectors of parameters were introduced by FADA. All these parameter vectors correspond to the single iteration vector (j_2, i_2) of the source function `readTarget()`. Therefore, at lines 9 and 11 only one local and one global control arrays are generated for this pair of functions. Similarly, at lines 43 and 45 one local and one global control arrays are added for the pair of functions `absVal()` and `vertSum()`.

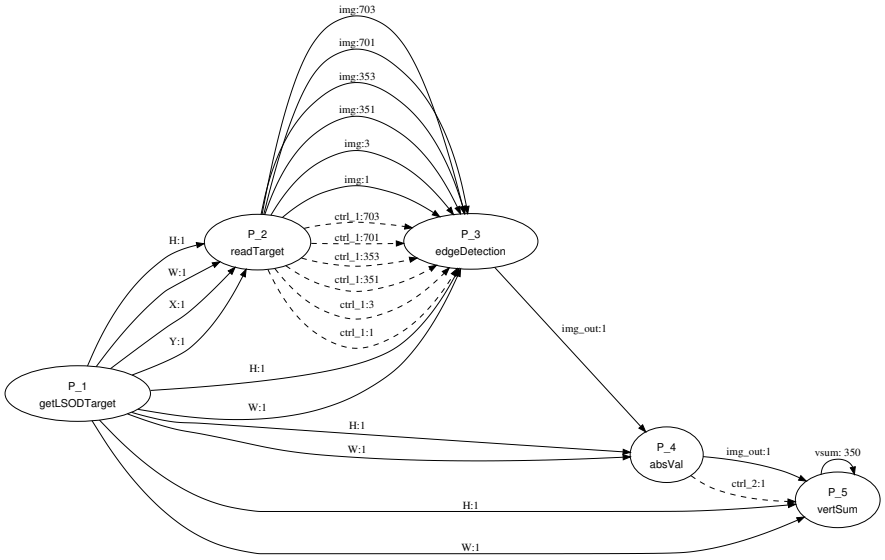


Figure 6.3: The PPN derived from the LSOD program.

By applying Step 4 (see Section 3.5) of our approach to the final dSAC of the LSOD application depicted in Figure 6.2, we derive the topology of the PPN depicted in Figure 6.3. The topology consists of 5 processes, 19 *data channels* shown as solid lines that are used to exchange data between processes and 7 *control channels* shown as dashed lines used to propagate values of global control arrays. The edges of the PPN in Figure 6.3 are annotated with the buffer sizes calculated for the LSOD program considering maximum dimensions of the targets to be 350x300 pixels. This means

that we set maxWidth = 350 and maxHeight = 300.

Finally, as an example of the internal structure of the PPN processes, in Figure 6.4, we present the pseudo code for readTarget and edgeDetection processes derived after the linearization step. These processes exhibit the most intensive data communication in the PPN. The internal code structures of these processes are generated as it has been explained in Step 4 (see Section 3.5) of our solution approach. Note, that the input/output ports used to access FIFO channels (see the read/write primitives in Figure 6.4) are automatically mapped to physical addresses generated by the Espam tool (in a separate header file).

```
1  for k = 1 to Targets,
2    for j = 0 to maxHeight + 1,
3      for i = 0 to maxWidth + 1,
4        if (j == 0 && i == 0)
5          read(0, H)
6          read(1, W)
7          read(2, X)
8          read(3, Y)
9        endif
19       if (j <= H + 1 && i <= W + 1)
11         img_1 = readTarget(X,Y)
12         lcl_1 = (j,i)
13         if( j <= H-1 && i <= W-1 )
14           write( 1, img_1 )
15         if( j <= H-1 && i >= 2 )
16           write( 3, img_1 )
           ...
17       endif
18       ctrl_1 = lcl_1
         if( j <= H-1 && i <= W-1 )
           write( 0, ctrl_1 )
         if( j <= H-1  && i >= 2 )
           write( 1, ctrl_1 )
         ...
23     endfor
24   endfor
25 endfor
```

(a) Process readTarget

```
1  for k = 1 to Targets,
2    for j = 1 to maxHeight,
3      for i = 1 to maxWidth,
4        if (j == 0 && i == 0)
5          read(0, H)
6          read(1, W)
7        endif
8        if (j <= H && i <= W)
9          read(0, ctrl_1)
10         if (ctrl_1.j == j-1 &&
               ctrl_1.i == i-1)
11           read(1, in_0)
12         read(2, ctrl_1)
13         if (ctrl_1.j == j-1 &&
               ctrl_1.i == i+1)
14           read(3, in_1)
           ...
15         img_out = edgeDetection(
                     in_0, in_1,
                     in_2, in_3,
                     in_4, in_5)
           write(0, img_out)
         endif
19     endfor
20   endfor
21 endfor
```

(b) Process edgeDetection

Figure 6.4: Internal code structures of processes readTarget and edgeDetection of the PPN derived from the LSOD application.

We evaluate our approach by executing the parallel LSOD application specification on an Intel® Xeon® quad-core machine running a Linux operating system. We used the ESPAM [55] tool and the HDPC [56] library to map the processes of the generated PPN onto cores and to generate C++ code for these cores. We used the GCC compiler to generate the final binary code. The HDPC library employs the boost-thread framework that enables the use of multi-threaded implementations. That is, the derived PPN has been translated to a multi-threaded program realizing the LSOD application, in which every process of the PPN is a separate thread.

In this experiment, we implemented and executed the parallel PPN specification of the LSOD application considering 5 different mapping configurations. The obtained results are shown in Figure 6.5. The horizontal axis depicts the number of cores used

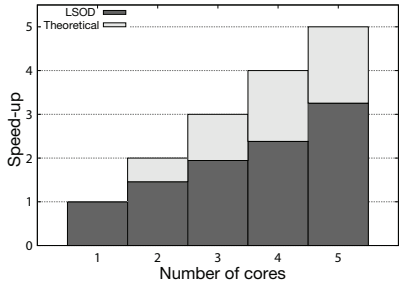


Figure 6.5: Evaluation of LSOD PPN on several CPUs.

in each configuration, i.e., we mapped the 5 processes of the PPN onto 1, 2, 3, 4, and 5 cores, respectively. Note that because the `Intel® Xeon®` processors support hyper-threading, the operating systems 'sees' 8 different cores. Therefore, we could map 5 processes on 5 different cores exploiting maximum concurrency. Obviously, when using less than 5 cores, some processes have to share the same core. In this case, we let the operating system to schedule the execution of these processes (i.e., threads) on a particular core. We experimented with grouping different processes together, i.e., mapping several processes on a single core. Figure 6.5 presents the best results that we have obtained. In addition, every configuration has been executed multiple times and the bars show the obtained average speed-up. The first configuration (see the leftmost bar in Figure 6.5) represents our reference configuration, in which, we mapped all 5 processes of the PPN onto a single core. We consider the speed-up of this configuration to be 1. Also, we have normalized the performance of the other configurations with respect to the performance of this reference configuration. Looking at the performance of the other 4 configurations, we see that by increasing the number of cores, the speed-up increases below the theoretical maximum shown as gray bars in Figure 6.5. We found that because of the data dependencies in the LSOD application and the imbalanced workload of the functions executed by different processes, the theoretical maximum cannot be achieved. In addition, in all configurations except the one using 5 cores (see the rightmost bar in Figure 6.5), there is an overhead introduced by the operating system for scheduling different threads on one core. As a result, the rightmost configuration exhibits a slightly larger speed-up increase compared to the other configurations. Finally, there is the execution time overhead caused by the extra 'dummy' iterations, which we discussed in Section 3.7. Below, we present some numbers with regards to this execution time overhead, as well as, the memory overhead in the LSOD process network.

Execution time overhead

From the execution statistics obtained by profiling of the LSOD application and its PPN, we computed the two ratios in Equation 3.4 presented in Section 3.7. Table 6.1 shows the ratios for each process and the whole PPN. Note that for computing $(max\_f - x)/x$, we need to consider that the targets are 2-dimensional. Therefore,

we used the term:

$$\frac{maxWidth \cdot maxHeight - x \cdot y}{x \cdot y} = \frac{350 \cdot 300 - x \cdot y}{x \cdot y}.$$

The terms x and y represent the average target size, which we computed from the targets used when executing the program. Based on the computed values in the last column of Table 6.1 and applying Equation 3.4 in Section 3.7, the overhead due to the execution of 'dummy' iterations of the LSOD PPN is 33.93%.

Process	P_1	P_2	P_3	P_4	P_5	PPN
$W/(W_x + W)$	0.53	0.38	0.26	0.47	0.3	0.39
$(max\_f - x)/x$	0	1.09	1.09	1.09	1.09	0.87

Table 6.1: Statistics LSOD PPN.

Memory overhead

In order to evaluate the memory overhead, we measured the memory requirements for the sequential LSOD program and compared this with the memory requirements for executing the corresponding PPN. The sequential program requires in total 13650 Bytes of memory, which includes both the code and the data. In order to make a fair comparison, it is important to note that this number (13650 Bytes) does not include the data array used to buffer the largest possible target, i.e., variable `img[TH][TW]` which requires 350x300 = 105000 Bytes. Although, we use such a variable in the sequential program, the program can be written more efficiently in a way that we do not need to buffer the whole (largest possible) target. The left part of Table 6.2 shows the memory requirements for every process in the generated process network. In addition, we need to consider also the memory used to implement the FIFO channels. In total, the PPN requires 17018 Bytes for implementing the processes and 18384 Bytes for the FIFO channels, see the right part of Figure 6.2. Then, if we compare these numbers with the number of the sequential program, we see that the memory overhead is 2.6x, which is reasonable provided that this is the memory requirement for the implementation of 5 processes and 26 FIFO channels.

Process	P_1	P_2	P_3	P_4	P_5
Code (bytes)	1626	2302	2510	1742	1978
Data (bytes)	1420	1360	1360	1360	1360
Total (bytes)	3046	3662	3870	3102	3338

	PPN	Sequential
Memory (bytes)	17018	13650
FIFOs (bytes)	18384	–
Overhead	2.6x	–

Table 6.2: Memory overhead of the LSOD PPN.

Overall, the average efficiency of the 4 parallel implementations of the LSOD process network is around 70%. The efficiency (Eff) is defined as:

$$Eff = \frac{SP}{C},$$

where SP is the speed-up and C represent the number of cores used to achieve this speed-up.

6.2 Discussion and Summary

In this chapter, we evaluated our parallelization approach presented in Chapter 3 on a real-life application called Low Speed Obstacle Detection (LSOD). This application contains for-loops with dynamic bounds and is an example of an application with relaxation II presented in Section 1.2. By evaluation of this application, we demonstrated the practical applicability of our parallelization approach to a real-life dynamic application.

By evaluating our approach we found that because of the data dependencies in the LSOD application and the imbalanced workload of the functions executed by different processes, the theoretical maximum cannot be achieved. In addition, there are two types of overhead in the generated PPN, i.e., memory and execution time overhead. The execution overhead is caused by the execution of some 'dummy' iterations not present in the initial sequential program. The memory overhead is due to the introduced control arrays, as well as, the created dataflow and control FIFO channels. It highly depends on the characteristics of the application being parallelized. In Section 3.7, we presented analytical analysis of the execution overhead. For the LSOD application, the overhead due to the execution of 'dummy' iterations of the LSOD PPN is 33.93%. This overhead is highly dependent on the maximum dimensions of the targets in the image.

In order to evaluate the memory overhead, we measured the memory requirements for the sequential LSOD program and compared this with the memory requirements for executing the corresponding PPN. The total memory overhead is 2.6x which is reasonable, because this is the memory requirement to implement 5 separate processes and 26 FIFO channels which is unavoidable if we want to parallelize the LSOD to the maximum task-level parallelism available in the initial LSOD specification. Overall, the average efficiency of the 4 parallel implementations of the LSOD process network is around 70% which is very reasonable for parallel implementation.

From the LSOD evaluation, the obtained results indicate that the approach we presented in Chapter 3 facilitates efficient parallel implementations of sequential nested loop programs with dynamic loop bounds. That is, our approach reveals the possible parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.