Deadlock detection in Kahn Process Networks

Erik Jongsma
LIACS, Leiden (ejongsma@liacs.nl)
Supervisor: Dr. H.N. Nikolov
LIACS, Leiden (nikolov@liacs.nl)

December 20, 2010

Contents

1 Introduction

One of the biggest challenges computer scientists face at this moment is parallel programming. Individual CPU core speeds cannot be increased much further, due to technical and physical restraints. Therefore, CPU developers have started to design to multi-core processors. The theoretical increase in performance that can be gained using this approach is given by Amdahl's Law:

$$\text{speedup}(p) = \frac{1}{s + (1-s)/p},$$

where p is the number of cores, and s is the fraction of the program in question that can not be parallelized (the serial part). Of course, this formula is a simplification of reality, as for example we could have programs which can be parallelized perfectly for two cores, but not more. Assuming we can find "nice" programs, in the sense that their s is small, we can get a big improvement in performance by using multiple cores.

However, most software that has been written so far is sequential, i.e., not programmed with parallel execution in mind. Combine this with the fact that parallel programming is:

- Hard for developers to do by hand,
- Very time-consuming,
- Very error-prone,

and it can be seen that there are challenges to be overcome in this area. The rest of this thesis is organized as follows: we start by talking about the background of our work. This includes sections about Kahn Process Networks, HDPC, etc. Then, we will discuss deadlock detection. A couple of small examples will be given, followed by a list of possible approaches to deadlock detection. We will then present the algorithm we used and the way it is implemented. Finally, we will give some results obtained by using our implementation on several programs.

2 Background

Programming for multiprocessor systems is a very difficult and time consuming process. Because of this, and the fact that multiprocessor systems have not been around that long, most existing applications have been written as sequential programs. Therefore, we would like to have a method to automatically convert these existing programs to parallel programs that are optimized for execution on multiprocessor platforms.

The LERC group at the LIACS has created a tool-flow to automate this process, called Daedalus:

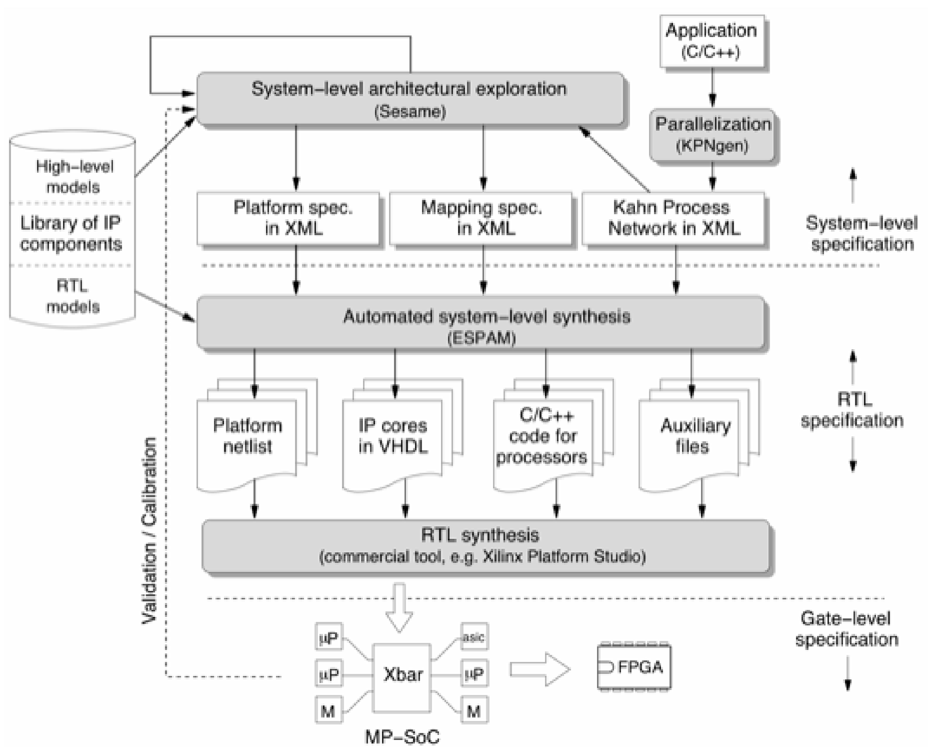


Figure 1: Daedalus tool flow.

The Daedalus design flow, depicted in Figure 1, provides a single environment for rapid system-level architectural exploration, system-level synthesis, programming and prototyping of multimedia MPSoC (multiprocessor platforms) architectures.

Here, a key assumption is that the MPSoCs (multiprocessor platforms) are constructed from a library of pre-determined and pre-verified IP components. These components include a variety of programmable and dedicated processors, memories and interconnects, thereby allowing the implementation of a wide range of embedded MPSoC platforms.

Starting from a sequential application specification in C, the PNgen tool (see Section 2.2) automatically converts it into a parallel Kahn Process Network (see Section 2.1) specification. To enable the automation, the input programs are restricted to so-called static affine nested loop programs, which are an important class of programs in, e.g., the scientific and multimedia application domains.

The generated KPNs are subsequently used by the Sesame modeling and simulation environment to perform system-level design space exploration (DSE). Sesame uses high-level model components from the IP component library, see the right part of Figure 1.

The DSE results in a number of promising candidate system designs. Their system-level specifications, i.e., platform, application-to-architecture mapping, and application descriptions, act as input to the ESPAM tool (see Section 2.3).

The ESPAM tool uses these system-level input specifications, together with RTL versions of the components from the IP library, to automatically generate synthesizable VHDL that implements the candidate MPSoC platform architecture. In addition, it also generates the *C* code for those application processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can be readily mapped onto an FPGA for prototyping.

However, instead of synthesizing these processes onto an FPGA, we will use the HDPC framework (see Section 2.4). This framework functions similar to the YAPI tool, used for fast simulation of process networks on desktop PCs. In contrast with YAPI, the HDPC framework does not use its own real-time environment. HDPC generates a cross-platform (Windows, Linux, Mac OS) multi-threaded implementation of KPNs targeting multi-core platforms. HDPC keeps the execution overhead of the generated KPN minimal. Therefore, it can be used not only for simulations but it can also be used as the final implementation if the target is general-purpose multi-core desktop machines.

2.1 Kahn Process Networks

The first step of the process described above is representing a sequential application in a parallel model of computation. The model we use is the Kahn Process Network (KPN), by Dr. Gilles Kahn [Kah74]. This model is especially suited for applications dominated by data-flow, e.g. streaming image manipulation.

A KPN consists of concurrent processes that communicate using unbounded First-In First-Out (FIFO) channels. Processes produce data tokens, which are then written to channels. These tokens will remain in the channels until they are consumed by other processes. Each channel has only one producer and one consumer process connected to it, so multiple producers or consumers per channel are not allowed in this model. If a process attempts to read from an empty channel, it will block until the corresponding process fills the channel with a new data token. If a process is running, it will only access one channel at a time, and if blocked, it will not be allowed to access other channels. Kahn Process Networks can be represented as directed graphs.

One of the nice things about the KPN model is that the result of its computation is independent of execution order, as long as we ensure that processes block when trying to read from an empty channel. This allows us to execute the network in parallel, as well as sequentially. However, the KPN model assumes channels that have unbounded capacity. As we will not have an unbounded amount of memory when implementing a KPN, we will have to use fixed channel sizes.

Therefore, instead of KPNs, we will use Polyhedral Process Networks (PPNs). PPNs are a special case of KPNs. In PPNs, the communication FIFO channels are finite and therefore, PPNs synchronize using both blocking read and blocking write, whereas KPNs only require blocking read. Also, in PPNs the processes are internally structured in a particular way. That is, each process executes three phases in a loop, namely the Read, Execute, and Write phases.

For example, we might have something like Figure 2:

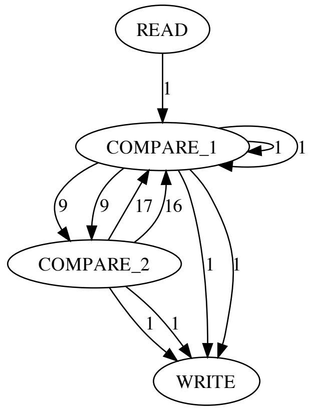


Figure 2: PPN Example

In this figure, we can see four processes and 11 channels. The numbers beside the channels represent the channel sizes.

The question is: how large should our channel sizes be? If we make them too small, we might run into a situation where our application is in a deadlock state (because if our channels are bounded, processes can also block on writing to a full channel). On the other hand, we would like our channels to be as small as possible, to minimize memory usage. We will discuss this problem in more detail in the upcoming sections.

2.2 PNgen

For most programs, manually specifying an application as a process network is very error-prone and time-consuming. Therefore, a tool called PNgen has been created [VNS07]. PNgen is a tool that takes a sequential C/C++ program as its input, and outputs a behaviorally equivalent PPN. The only restriction on the C/C++ code is that the program must be a static affine nested loop program (SANLP).

A SANLP consists of a set of statements, each possibly enclosed in loops and/or guarded by conditions. The loops need not be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses can contain enclosing loop iterators and parameters as well as modulo and integer divisions. The parameters are symbolic constants, that is, their values may not change during the execution of the program fragment.

SANLPs are common in scientific computing (e.g., matrix computation) and signal processing applications. The reason for restricting to these programs is that the dependence analysis necessary to derive the channels may not be possible in general, but it is possible for SANLPs.

PNgen also generates channel sizes that ensure deadlock-free execution of the network. For self-loops, this computation is easy because it does not depend on the scheduling of the network. For other channels, a different approach is needed. The way PNgen computes these channel sizes is as follows: first, it computes a deadlock-free global schedule of the PPN. Then, the individual channel sizes are computed for this schedule. Note that the computed schedule may not be optimal and that the buffer sizes may not be valid for an optimal schedule. However, using this method ensures us that there is in fact a valid schedule for the computed buffer sizes. Based on the global schedule, all processes are placed in a common iteration space, forming one big compound process. As a result, all channels become self-loops. Therefore, we can then compute all channel sizes.

2.3 Espam

After we have used PNgen to generate a PPN, we must then use it to create a parallel implementation of our program. This is where we use the ESPAM (Embedded System-level Platform synthesis and Application Mapping) tool [NSD08]. ESPAM takes as input a system-level specification, consisting of three parts:

1. A Platform Specification, describing the topology of a platform using generic parameterized system components taken from a library (see Figure 1);

2. An Application Specification, describing an application as a PPN. The

PPN specification reveals the task-level parallelism available in the application;

3. A Mapping Specification, describing the relation between all processes and FIFO channels in Application Specification and all components in Platform Specification.

ESPAM can target different platforms on which the PPN will be executed (YAPI, System C, FPGA, etc.)

2.4 HDPC

The Heterogeneous Desktop Parallel Computing Framework (HDPC) [Far08] is another target for ESPAM. For this target, ESPAM generates backend code for a desktop computer that acts as the controlling and coordinating arbiter between the processes of a PPN. The processes can then execute on various computing devices like the FPGA, Graphics Processing Unit (GPU), or the Cell B.E. to take advantage of their respective strengths. For each process in a PPN, a thread on the host CPU is created.

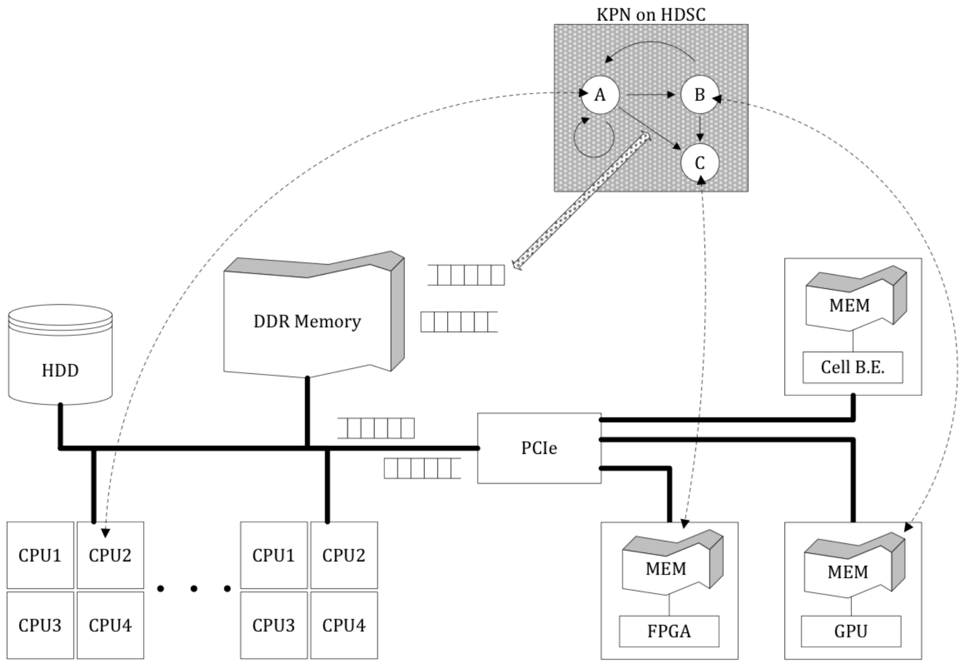


Figure 3: HDPC Framework

Figure 3 visualizes the approach; a PPN running on the HDPC framework with three interconnected processes. A, B, and C all execute their functions on a device connected to the same machine. Communication channels and the FIFO mechanism are implemented in main system memory and are under HDPC's control. This simplifies memory management, as we are working in the same address space.

As the framework acts as a controlling and coordinating framework of a PPN, all on a single machine, there are several differences to the traditional task-parallel approach. PPNs for example do not allow, or even consider, global variables for communication between processes. As in HDPC all communication happens in the same shared-memory system, the use of global variables is permitted. These can be used for example as read-only values for control, constants, etc.

3 Deadlock detection

In this thesis, we are adding real-time deadlock detection and dynamic channel resizing to the HDPC framework. In theory, this would not be necessary, because if we use PNgen we get channel sizes that guarantee deadlock-free execution. However, there are several reasons why we would want this anyway:

- The channel sizes given by PNgen are not always minimal. Using deadlock detection (and dynamic resizing of channels) at runtime allows us to find the absolute minimum channel sizes for the actual execution schedule, thus minimizing memory usage.
- If we design a program by hand in the HDPC framework, there are no guarantees on the buffer sizes at all. So, in this case, we do require a mechanism to deal with deadlocks.
- In some cases, the minimal channel sizes might be data-dependent. When using a run-time deadlock detection algorithm, we will always get the correct minimal sizes.

Two kinds of deadlocks can occur in process networks: artificial deadlocks and real deadlocks. Real deadlocks are deadlocks which are independent of channel sizes. This means all channels involved in a real deadlock will be blocked on read. Artificial deadlocks are deadlocks which only occur because we do not have infinite channel sizes as required by the definition of a KPN. They can be resolved by increasing the amount of memory reserved for the channel causing the deadlock. Every deadlock which involves at least one channel blocked on write is an artificial deadlock.

We will first show some examples of situations in which deadlocks can occur. Then, we will briefly describe the different methods that have been used in the past to resolve deadlocks. Finally, we will discuss our method.

3.1 Examples

First, a simple example where artificial deadlock will occur:

3.1.1 Artificial deadlock

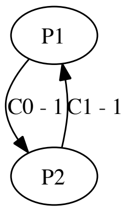


Figure 4: PPN with an artificial deadlock

P1 and P2 both do the same in a loop: they write two tokens to their output channel, and then read two tokens from their input channel. When we execute this PPN, both channels will be filled and then we are in a deadlock state. Both processes still need to write one more token to their output channel, but those are both full. This is an artificial deadlock, because if the channel sizes were bigger, we would not have this problem. Indeed, if this PPN is executed using our deadlock detection and resolving algorithm, the size of one of the channels is increased to two, and the program can continue.

3.1.2 Real deadlock

Now, a simple example to show when real deadlocks occur. We look at the same KPN:

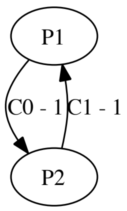


Figure 5: PPN with a real deadlock

P1 and P2 still do the same in a loop: they read two tokens from their input channel, and then write two tokens to their output channel. When we execute this PPN, both channels will be empty at the start, and so both processes will not be able to read. We are now also in a deadlock state, except that this deadlock cannot be resolved by increasing channel sizes. Therefore, this network is in a real deadlock.

3.1.3 Multiple deadlocks

In this example, multiple cycles can be found:

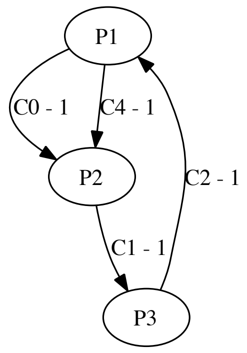


Figure 6: KPN with a multiple deadlocks

In this example, we will not specify what the different processes do, but just show some of the possible cycles:

- C1 → C2 → C4, all blocked on write: artificial deadlock.
- C1 → C2 → C4, all blocked on read: real deadlock.
- C0 → C4, with one of the two blocked on read: artificial deadlock.

More possible cycles can be found in this graph. Note that when looking for deadlock cycles, we reverse the direction of the edges if a channel is blocked on read (as in the last case). This will be explained in more detail when we discuss the deadlock detection algorithm.

3.2 Other approaches

A lot of effort has gone into deadlock detection for process networks, leading to all kinds of solutions to this problem. For some languages, like SHIM (a concurrent language), this problem is solved statically [SVE09]. However, this gives us no result about channel sizes, it just detects if the current configuration will lead to a deadlock state when executed. Then, there are several approaches that deal with process networks. W. Huang and D. Qi [HQ08] use message passing between processes. They communicate with each other about their status, and with the so-called resource managers (the channels). Deadlocks are detected by the resource managers, when they have had messages from all processes involved in it. N. Barath et. al. [BNB05] use a different approach. They use a Deadlock Detection and Resolution (DD&R) process, which is pre-programmed with all possible cycles in the process network. This process then monitors the other processes at run-time, and detects a deadlock when one of these cycles is found. Others [OE05, AZE07] detect deadlocks in a distributed way. TCP/IP messages are used to send labels to other processes. These labels keep track of the state of the processes. Once all processes in a cycle have the same label, a deadlock is detected. B. Jiang et. al. [JDK08] propose a hierarchical deadlock detection algorithm, where the process network is split into segments, and each segment gets its own local observer. A global observer is then used to gather data from all local observers and detect a deadlock if it occurs.

3.3 Our work

Instead of the different processes communicating with each other about their status, we have used another approach. Separate from the threads executing the processes of the PPN, we have a watcher thread. We will first discuss the way our deadlock detection algorithm works. Then, we will look at the modifications have been made to the HDPC framework. Finally, we will discuss the watcher thread and dynamic resizing.

The method we use for deadlock detection is similar to the one used by Jiang et. al. [JDK08], without the hierarchical part. As we have seen before, a PPN can be represented as a graph, where the vertices are processes and the edges are channels. Using our watcher thread, we can get blocking information about this graph. So, we know which of the processes are blocked on read or write (or not at all). Using this information, and the graph layout, we construct a dependency graph. This is done as follows: if none of the processes attached to a channel are blocked, it is removed from the graph. If a process is blocked on read, the direction of the corresponding channel is reversed. This is because if a process is blocked on read, it is dependent on the writing process. If a process is blocked on write, the corresponding channel remains as is. The dependency graph represents a state of the program, where the edges are pointing from blocked processes to the processes they are waiting for.

To clarify this, we look at an example from [JDK08]. Suppose we have the following PPN:

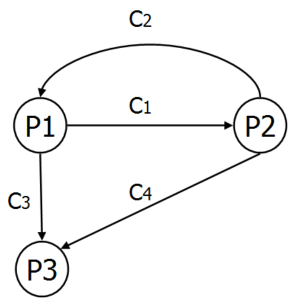


Figure 7: A PPN

Also suppose our watcher thread has determined that P_1 and P_2 are blocked on write, and that P_3 is blocked on read. Then, C_2 is removed, the direction of C_3 is swapped, and we get the following dependency graph:

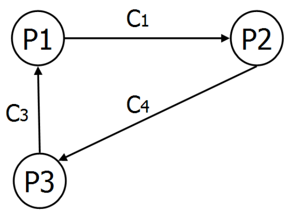


Figure 8: A dependency graph

Now we can detect deadlocks by looking for cycles in the dependency graph. Indeed, if there is a cycle in de dependency graph, we know that there is a number of processes which are all waiting on each other. In the example, we have a cycle consisting of C_1, C_4 and C_3.

Once we have found a cycle, we first check which kind of deadlock state the program is in. If a cycle consists of only processes blocked on read, we have found a real deadlock. We output this and quit the program. Otherwise, we have an artificial deadlock, and we want to increase the size of the channel causing it. To do this, we use timestamps. Each time the status of a channel is changed by one of its processes (blocked on read/write, or not blocked), we store a timestamp recording when this happened. Once a deadlock is detected, we increase the size of the channel which has the smallest timestamp. Of course, we only increase the size of full channel, so empty channels are disregarded. After increasing the channel size, we allow execution to continue. We keep using this method until the program finishes, and finally output the channel sizes we have found.

Now, we look at the modification made to the HDPC framework. We started by moving the initialization of the channels. Before, channels were created in a method of the PPN processes called `attachinput`. However, since the watcher thread needs access to the channels, they are now created in the main thread of the program, so that a list of channel pointers can be made. The `attachinput` method of the processes has also been changed, to allow it to bind the processes to already existing channels. Furthermore, we need to allow the watcher access to the PPN topology. So, we have created the following data-structure:

```
typedef struct ChannelData {
    int Number;
    int Input;
    int Output;
    ChannelBase* ChannelPointer;
    int Status;
    uint64_t Blocktime;
} ChannelData;
```

One instance of this struct is created in the main thread. The `number` variable is simply the channel name (used for output). The rest are fairly straightforward, `ChannelPointer` is the pointer to the actual channel, and the last two variables are only used by the watcher thread and will be discussed later. To see how we must adapt an existing HDPC project to work with this new framework, refer to section 4.1.

Then, we changed all channel types. For example, the `wait_read` function of the semaphore type channel is now:

```
#if HDPC_DEBUG_MODE
    bool done = false;
    while (!done) {
        Write.lock();
        if (!full.try_wait()) {
            if ( Status == 0 )
                Timestamp = getTime();
            Status = 1;
        } else
            done = true;
        Write.unlock();
        boost::this_thread::yield();
    }
    Read.lock();
    Status = 0;
#else
    wait (full);
#endif
```

The same changes have been made to all channel types. The main point is that once the channel is empty (and a process is blocking on it), `Status` is set to 1 and `Timestamp` to the current time. Until the status changes, the timestamp will remain the same. For the `wait_write` method, the changes are almost the same, except the status will be set to 2 when the attached process is blocked on write. Furthermore, some mutexes have been introduced, which we will discuss in more detail later on.

Now, we are ready the discuss the watcher thread. This thread executes the following steps in a loop:

- All current read and write operations are finished, and no further operations are allowed to start. We use the `Read` and `Write` mutexes mentioned earlier to facilitate this. So, the watcher simply locks both mutexes for all channels.

- Status information is collected in the form of block status (full, empty, not blocked), and the time of the latest change to this status. The watcher uses methods of the channel to get the `Status` and `Timestamp` values.

- Read and write operations can be resumed. Methods of the channel are used to unlock both mutexes for all channels.

- We calculate if a deadlock has occurred:

  First, all blocked channels are sorted by block-time. This is done by a simple sorting algorithm.

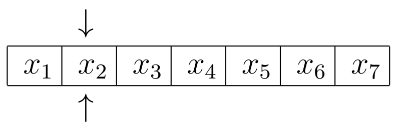
  Then, for each blocked channel, we determine if there is a cycle in the dependency graph using channels that are blocked after it (in time). If so, we have found a deadlock.

  If a real deadlock has been found, we output it and exit the program.

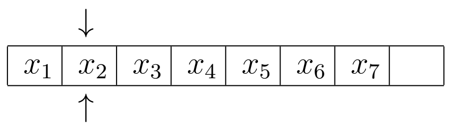
  If an artificial deadlock has been found, we increase the size of the channel causing it (as determined by the timestamps), and continue executing the program.

Because in a PPN a process can be blocked on only one channel, the cycle-finding algorithm is very efficient. Also, because the watcher process synchronizes the entire network before checking its status, we can immediately detect deadlocks without finding false positives. Other approaches require polling methods to determine if a deadlock is a false positive, leading to difficulties such as the need to determine polling intervals (see for example [JDK08]).

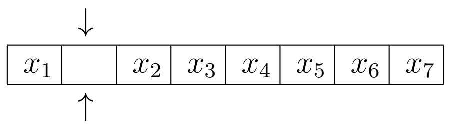
The dynamic resizing of the channels is done as follows: At the time a channel is full, the read and write pointer will point to the same channel element, like this:



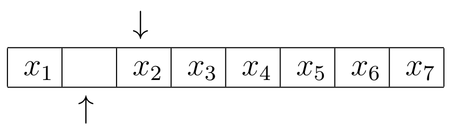
We start by increasing the size of the channel in memory. The old contents of the channel are automatically copied:



Then, we move the contents of the channel after the read-pointer one place forward:



Finally, we move the read pointer one place forward. We now have space in the channel for one extra element, and the order in which elements will be read will stay the same:



3.3.1 Determining minimal buffer sizes

Because we can dynamically resize channels, we can start our program with all channel sizes set to 1. During execution, channel sizes are then increased as required until the program can finish execution. This guarantees that we have actually found the minimal channel sizes for the actual execution schedule. We will see in the result section that the difference between the channel sizes generated by PNgen and our approach is sometimes quite large.

4 Results

4.1 Getting Started

First, we will describe how to change a HDPC program such that it works with our new framework which includes deadlock detection and resizing. Hopefully, these changes will be added to ESPAM, so that this will not have to be done by hand. The code we will change here is a simple example, but the same method is used for all other results as well.

We will reuse the deadlock example shown earlier:

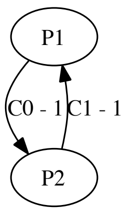


Figure 9: Example PPN

We will only edit the `<programname>_KPN.cpp` file. First, we add

```
#include <hdpc/channels/channel.h>

#if defined HDPC_DEBUG
#include <hdpc/watcher.hpp>
#endif
```

to the top of the file. This includes the watcher file if debugging (and thus, deadlock detection and resolving) is turned on. Also, we need the channel class definition regardless of debug state. Then, we look for the following code:

```
p_P_1.attachinput<LOCK_FREE, tCH_1>( 0, p_P_2.getOutPort(0), 1 );
p_P_2.attachinput<LOCK_FREE, tCH_2>( 0, p_P_1.getOutPort(0), 1 );
```

Here, a member function of each process is called to create a channel and attach its in- and outputs. As we now have to create the channels in the main thread, we comment the entire block and replace it by this:

```
using namespace hdpc;
using namespace channel;

// Create channels
ChannelBase* Channels[2];
Channels[0] = new Channel<tCH_1, LOCK_FREE>(1);
Channels[1] = new Channel<tCH_2, LOCK_FREE>(1);

p_P_1.attachinput( 0, p_P_2.getOutPort(0), Channels[0] );
p_P_2.attachinput( 0, p_P_1.getOutPort(0), Channels[1] );
```

Of course, if the program has more channels, the channel array defined above must be increased in size. Now, we need to generate a data structure for the PPN topology, so that the watcher can detect cycles. This is done as follows (these lines are added below the previous):

```
#ifdef HDPC_DEBUG
  // Create table for watcher thread.
  ChannelData Channelarray[2];
  WatcherData* Data = new WatcherData;
  Data->Channels = Channelarray;
  Data->NumChannels = 2;

  for (int i = 0; i < Data->NumChannels; i++) {
    Channelarray[i].ChannelPointer = Channels[i];
    Channelarray[i].Number = i;
  }

  Channelarray[0].Output = 1;
  Channelarray[1].Output = 2;

  Channelarray[0].Input = 2;
  Channelarray[1].Input = 1;

  // Create watcher thread.
  boost::thread* WatcherThread;

  // Start running the watcher.
  WatcherThread = new boost::thread(Watcher, Data);
#endif
```

This ensures the watcher has the PPN topology, and that it is actually started when debugging is enabled. Finally, we want to write the channel sizes we have found to a file, once we are done executing. So, we look for the following lines at the end of the file:

```
  tg.join_all();

  t.end_timer();
  printf("\nTime Elapsed: %.3f\n", t.elapsed_time());

  return 0;
```

and replace it by this:

```
  tg.join_all();

#ifdef HDPC_DEBUG
```

```
  WatcherThread->interrupt();

  t.end_timer();
  printf("\nTime Elapsed: %.3f\n", t.elapsed_time());

  ofstream output("ChannelSizesWorking.txt");
  for (int i = 0; i < 3; i++)
    output << "Channel " << i << ": " << Channels[i]->get_length() << endl;
#endif

  exit(0);
```

Again, the number above might need to be changed if there are more or less channels in the program. The watcher thread is killed after a normal execution of the program is completed. Then, the correct channel sizes are printed to the file `ChannelSizesWorking.txt`. These steps will ensure the program runs correctly using the modified HDPC framework. If debugging is enabled, all channel sizes can be set to 1, and the minimal required channel sizes will be stored. This procedure was used to generate the results in the following sections.

4.2 Odd-even sort

Odd-even sort is a sorting algorithm designed specifically for parallel execution. It works by passes of pairwise comparisons, similar to bubblesort. However, instead of one pass that is continuously repeated, we now have two passes. In pass 1, we compare each array element i with $i+1$ for all $i \equiv 0 \mod 2$. In pass 2, we compare each array element i with $i+1$ for all $i \equiv 1 \mod 2$. We continue executing these two passes after each other until the array is sorted. See below for the PPN of the algorithm, and the results. In the PPN, P1 is used to read elements from an array, P4 is used for output, and P2 and P3 are used to compare and swap values.

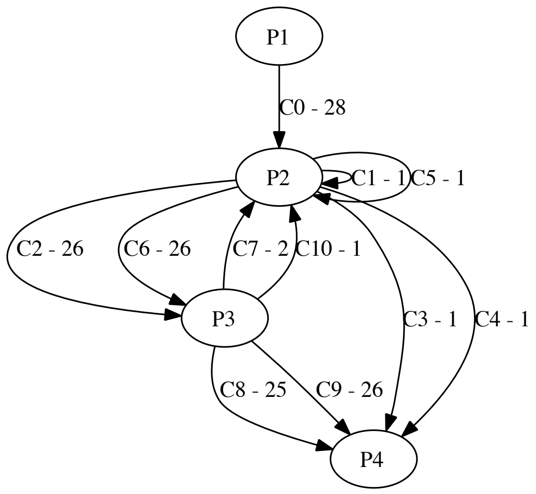


Figure 10: KPN for Odd-even sort

Channel	Original size	Determined size
0	28	1
1	1	1
2	26	9
3	1	1
4	1	1
5	1	1
6	26	9
7	2	17
8	25	1
9	26	1
10	1	16
Sum	**138**	**58**

Table 1: Buffer sizes determined at run time for Odd-even sort

The results of this test are quite interesting. We see that the channels from the input-process and to the output-process can actually be reduced to size 1 (C0, C8, C9). If we look at the channels between P2 and P3 (C2, C6, C7, C10), the difference is not as large. The sum of the original sizes for those is 55, while the determined size is 51. This small difference has to do with the fact that the processes store values in internal variables as well as in the channels. Hardware cores implementing processes may not have local memory, therefore the sizes given by PNgen are slightly larger.

The question is: does having smaller channels decrease performance because of excessive blocking? We have ran this example 20 times using both sets of channel sizes (with debugging disabled). The version with original channel sizes executed in 5.8 ms on average, while the minimal version took 7.9 ms. So, more blocking does slow down execution (as expected). However, it is up to the developer to decide whether to emphasize on speed or memory usage. These results give him/her this opportunity.

4.3 Sobel edge detection

Sobel is an edge detection algorithm where a 3×3 window is slid over the image to calculate the gradient of the pixel with its neighbours. In the PPN, P5 is used to read pixel values from the original, P1 is used for output to a new picture, P2 and P3 are used to compute the gradient per pixel, and P4 used to calculate absolute values.

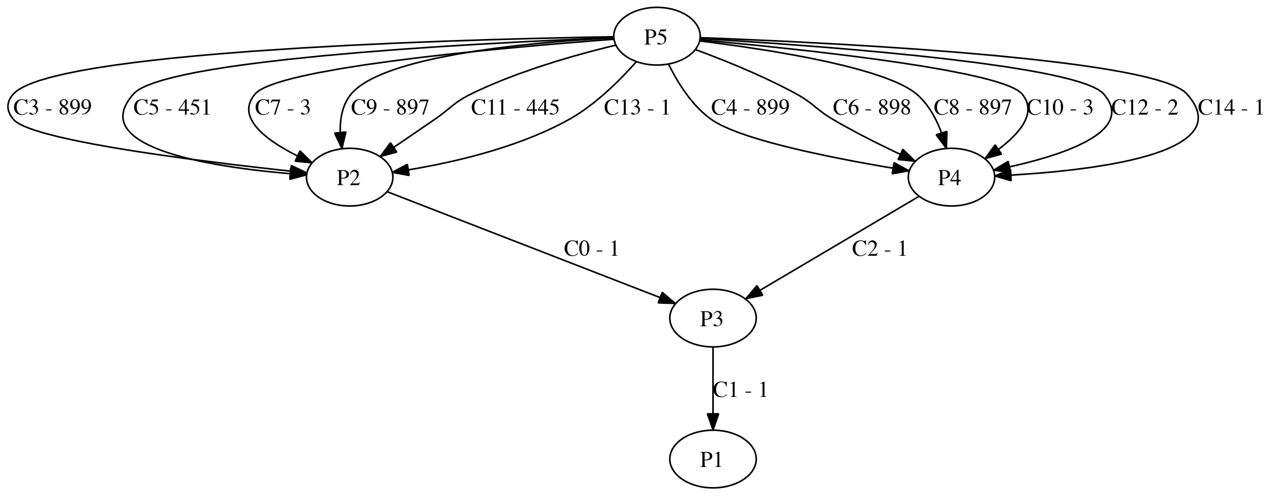


Figure 11: KPN for Sobel edge detection

Channel	Original size	Determined size
0	1	1
1	1	1
2	1	1
3	899	898
4	899	898
5	451	450
6	898	897
7	3	2
8	897	896
9	897	896
10	3	2
11	449	448
12	2	1
13	1	1
14	1	1
Sum	**5403**	**5393**

Table 2: Buffer sizes determined at run time for Sobel edge detection

Here, the results are not as interesting as in the odd-even test. However, these numbers do give us confidence that our approach is working correctly. The small differences between the original and the determined sizes are again because of the internal variables of the process. There are no significant differences in execution times for this example either.

4.4 Demosaic

Demosaic is an algorithm that takes raw pixel data from an image sensor (for example, a digital camera) and turns it into an RGB image. Because the PPN for this example is rather large, it can be found in appendix A.

Channel	Original size	Determined size	Channel	Original size	Determined size
0	1	1	27	62	60
1	1	1	28	1	1
2	1	1	29	1	1
3	259	257	30	63	62
4	1	39	31	63	61
5	1	68	32	2	2
6	1	1	33	62	1
7	62	61	34	2	1
8	62	61	35	65	65
9	63	61	36	258	127
10	2	1	37	62	61
11	259	132	38	1	1
12	65	65	39	1	1
13	1	1	40	1	1
14	2	1	41	61	61
15	1	1	42	1	1
16	1	1	43	1	1
17	62	61	44	2	1
18	1	1	45	62	61
19	1	1	46	62	62
20	61	61	47	1	1
21	1	1	48	62	59
22	1	1	49	62	62
23	62	62	50	1	1
24	1	1	51	1	26
25	1	1	52	1	1
26	62	62	**Sum**	**1997**	**1787**

Table 3: Buffer sizes determined at run time for Demosaic

In this example, we have some more interesting sizes. We will not talk about the differences of one or two, as these are because of the internal variables. Instead, we will focus on the main differences. We see that C4 and C5 are actually bigger than in the original case. C4 leads to P4 and C5 leads to P9 in the PPN. These both have outgoing channels that are a lot smaller in the run time case: C11 and C36, both going to P9. So, our hypothesis is that the data is "stored" earlier in the PPN, allowing for much smaller channels later on. Something similar is going on with C33 and C51. Again, this is probably because redistribution of weights to other edges of the graph.

With the determined sizes, the average execution time of the example is about 330 ms. With the original sizes (C17 set to 61), the average is 295 ms. As in odd-even, there is a choice to be made between speed and memory usage.

4.5 FDWT

FDWT is an algorithm that computes the forward discrete wavelet transform. This is a transformation that can be applied to images and sounds to get a series of coëfficients in a certain basis. The corresponding PPN can be found in appendix B.

Channel	Original size	Determined size	Channel	Original size	Determined size
0	1	1	17	1	1
1	450	449	18	1	1
2	901	900	19	1	1
3	451	449	20	112	1
4	450	450	21	112	1
5	1	1	22	1	1
6	450	450	23	113	1
7	2	1	24	1	1
8	1	1	25	1	1
9	3	1	26	1	1
10	113	1	27	1	1
11	1	1	28	112	1
12	1	1	29	112	1
13	1	1	30	1	1
14	3	1	31	113	1
15	1	1	32	1	1
16	113	1	**Sum**	**3627**	**2726**

Table 4: Buffer sizes determined at run time for FDWT

Apart from the usual one or two token difference, we can see that every channel that has capacity 112 or 113 is now reduced to one. As we can see in the PPN, these channels are all on a path to either P9 or P13. We think these channels are allowed to have size one, because these two processes can immediately pass on the received tokens to the next process.

Again, we compare the average execution time of the different versions. In this case, the minimal version runs about 2.20 s, while the original version runs in 1.53 s. Again, we see the same performance hit because of additional blocking.

5 Conclusion

From our tests, it seems the deadlock detection and dynamic resizing are working well. It even provided a significant reduction in memory used in several test-cases. However, as we can see from the execution times, the program invariably slows down when we decrease the channel sizes. Regardless, we think providing the absolute minimum channel sizes gives the developer the opportunity to make the trade-off between memory usage and speed. It could be that there are memory configurations in between the original and minimal one, which do not cause a performance decrease but are still more memory-efficient.

Further research on deadlock detection in the HDPC framework could be done. For example, we could investigate how often we want the watcher thread to execute its cycle. It might make sense to check less often if a program rarely reaches a deadlock state. This could even be done dynamically (if no deadlock is detected, sleep longer than the last time).

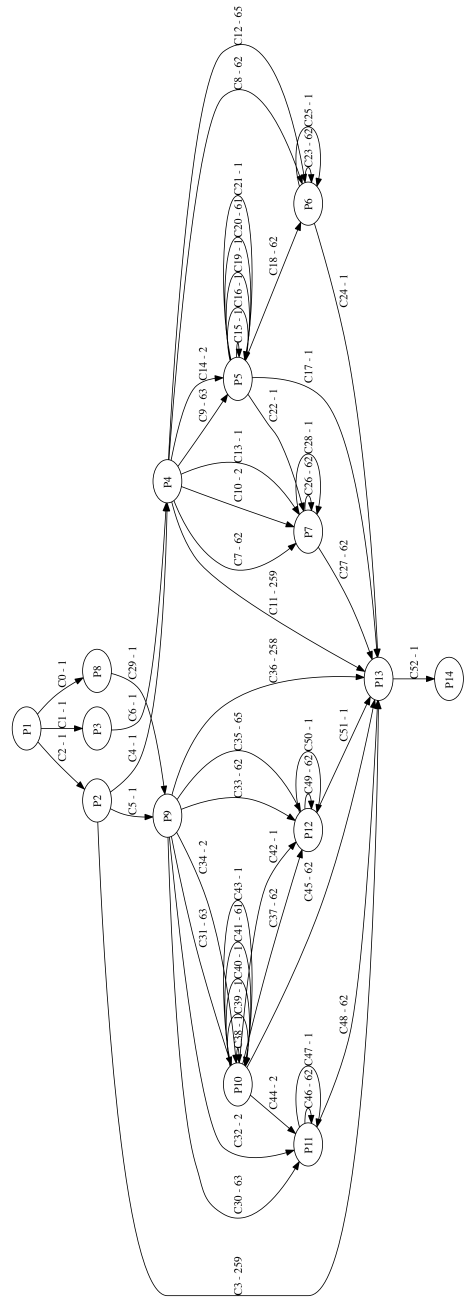
Another possible improvement could be varying the sizes of the channel increases. Currently, when a channel is increased in size, we create memory for one more token. If a channel requires a large amount of memory, the current implementation will take very long to figure this out. We could use a similar approach as the above, for example, we could increase a channel size by more than one if it has been the cause of a deadlock multiple times.

Finally, it would be interesting to test these examples on machines with many cores. It might be that the performance difference between the channel sizes determined by PNgen and our approach will be less if every process has its own processing core (because of less context switching when processes block).

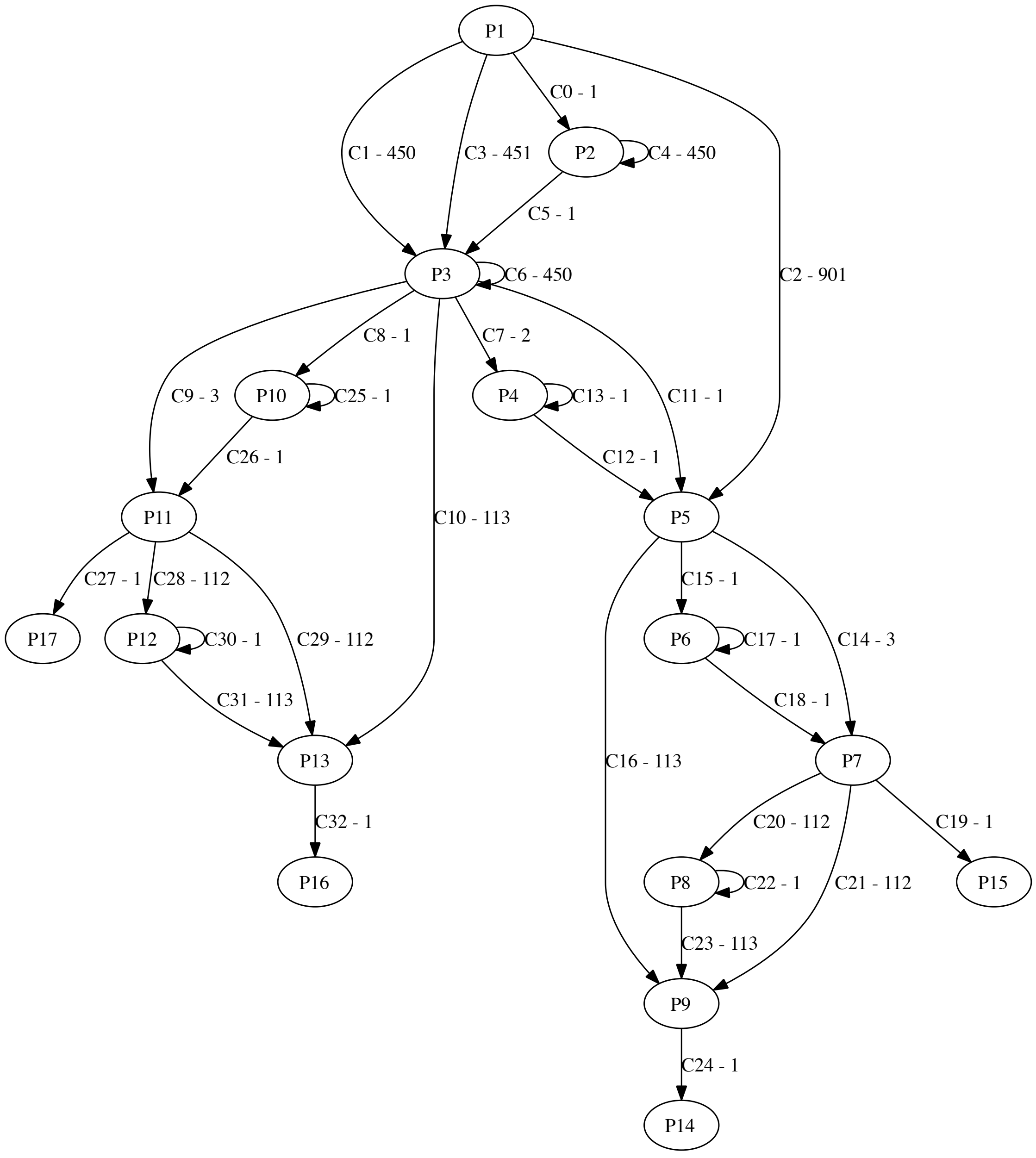
6 Acknowledgements

I would like to thank Hristo Nikolov for the inspiring meetings and helpful comments. I would also like to thank Tamás Faragó for his work on making the HDPC framework cross-platform, and the discussions about dynamic resizing.

A Demosaic KPN



B FDWT KPN


P1
C0 - 1
C1 - 450
C3 - 451
P2
C4 - 450
C5 - 1
C2 - 901
P3
C6 - 450
C8 - 1
C7 - 2
C9 - 3
P10
C25 - 1
P4
C13 - 1
C11 - 1
C26 - 1
C12 - 1
P11
C10 - 113
P5
C27 - 1
C28 - 112
C15 - 1
P17
P12
C30 - 1
C29 - 112
P6
C17 - 1
C14 - 3
C31 - 113
C18 - 1
P13
C16 - 113
P7
C32 - 1
C20 - 112
C19 - 1
P16
P8
C22 - 1
C21 - 112
P15
C23 - 113
P9
C24 - 1
P14

References

[AZE07] G.E. Allen, P.E. Zucknick, and B. L. Evans. A distributed deadlock detection and resolution algorithm for process networks. In *International Conference on Acoustics, Speech, and Signal Processing*, 2007.

[BNB05] N. Barath, S.K. Nandy, and N. Bussa. Artificial deadlock detection in process networks for eclipse. In *International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)*, 2005.

[Far08] T. Faragó. A framework for heterogeneous desktop parallel computing. Master's thesis, LIACS, Leiden University, 2008.

[HQ08] W. Huang and D. Qi. A local deadlock detection and resolution algorithm for process networks. In *International Conference on Computer Science and Software Engineering*, 2008.

[JDK08] B. Jiang, E. Deprettere, and B. Kienhuis. Hierarchical run time deadlock detection in process networks. In *IEEE Workshop on Signal Processing Systems*, 2008.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[NSD08] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.

[OE05] A.G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *International Conference on Acoustics, Speech, and Signal Processing*, 2005.

[SVE09] B. Shao, N. Vasudevan, and S.A. Edwards. Compositional deadlock detection for rendezvous communication. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 59–66, New York, NY, USA, 2009. ACM.

[VNS07] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems*, 2007.