HALCON

a product of MVTec

Parallel Programming (HDevelop)

HALCON 25.11 *Progress*

MVTec Software GmbH

Building Vision for Business

This technical note describes concepts for parallel programming with HDevelop, Version 25.11.0.0.

More information about HALCON can be found at: https://www.halcon.com

About This Technical Note

This technical note provides a practical starting point for a first look at parallel programming within HDevelop. It introduces you to different parallelization concepts and provides links to the detailed descriptions of several issues related to parallel programming within other documents like the HDevelop User's Guide and the Programmer's Guide.

This technical note is divided into the following parts:

- **Introduction**
  A short introduction to the meaning of "parallelization" within HALCON.
- **Parallelization Concepts**
  A description of common parallelization concepts, sorted by the resulting benefit, which can be the minimization of the response time, the maximization of the throughput, or the enhancement of the responsiveness.
- **Where to Find More Information**
  A guide to further parallelization related information contained in the HALCON documentation.
- **Glossary**
  A glossary for the most important terms related to parallelization.

Symbols

The following symbol is used within the manual:

! This symbol indicates an information you should **pay attention** to.

Contents

Chapter 1

Introduction

HALCON provides automatic operator parallelization (AOP) on the one hand and means for manual parallel programming of application parts on the other hand.

The automatic operator parallelization (AOP) splits the input data (e.g., an image) into parts and processes these data parts independently and in parallel. This is also called data parallelization and is described in section 2.1.1. By default, AOP is activated, i.e., this type of parallelization is done automatically and in many cases you don't have to care about further data parallelization, at least for individual operators. Details about AOP can be found in the Programmer's Guide, section 2.1 on page 15.

Manual parallelization is more complex and thus needs some more expert knowledge. This technical note describes common concepts of parallelization (chapter 2) that depend on the goal of the parallelization.

Further information related to parallel programming can be found all over the HALCON documentation. A guide to these information is provided with chapter 3 on page 25.

Finally, a glossary explains the most important terms related to parallelization (chapter 4 on page 27).

Chapter 2

Parallelization Concepts

Parallelization can be performed by several means, depending on the specific application and the goal of the parallelization. Here, some basic parallelization concepts are introduced. Only a selection of the most common general concepts can be provided. In the following, the parallelization concepts are related to the specific goals of the parallelization:

- to minimize the response time (section 2.1),
- to maximize the throughput (section 2.2 on page 13), and
- to enhance the responsiveness (section 2.3 on page 22)

of your application. However, practically you will often want to find a compromise between different goals.

2.1 Minimize Response Time

Basic concepts to minimize the response time of an application are data parallelization (section 2.1.1) and task parallelization (section 2.1.2 on page 12).

2.1.1 Data Parallelization and Automatic Operator Parallelization (AOP)

Data parallelization means that multiple threads perform the same task on different pieces of data.

The concept of data parallelization is shown in figure 2.1.

First, the data is split into several data pieces of approximately the same size. Then, the task is performed for each data piece individually by different threads. Finally, after the last thread has finished, the results of all threads are joined.

! By default, almost all operators automatically perform a data parallelization. **Note that in many cases, HALCON's automatic operator parallelization (AOP) is efficient enough so that a manual implementation of data parallelization does not lead to any further runtime enhancement.**

To explain the concept, in the following, an example for the manual implementation of data parallelization for a median filter is described.

2.1.1.1 Example: Simulate AOP

The HDevelop example program `hdevelop/System/Parallelization/simulate_aop.hdev` shows how to implement data parallelization for a median filter (see also figure 2.1) and compares the runtime with the runtime needed for the sequential processing and for HALCON's AOP.

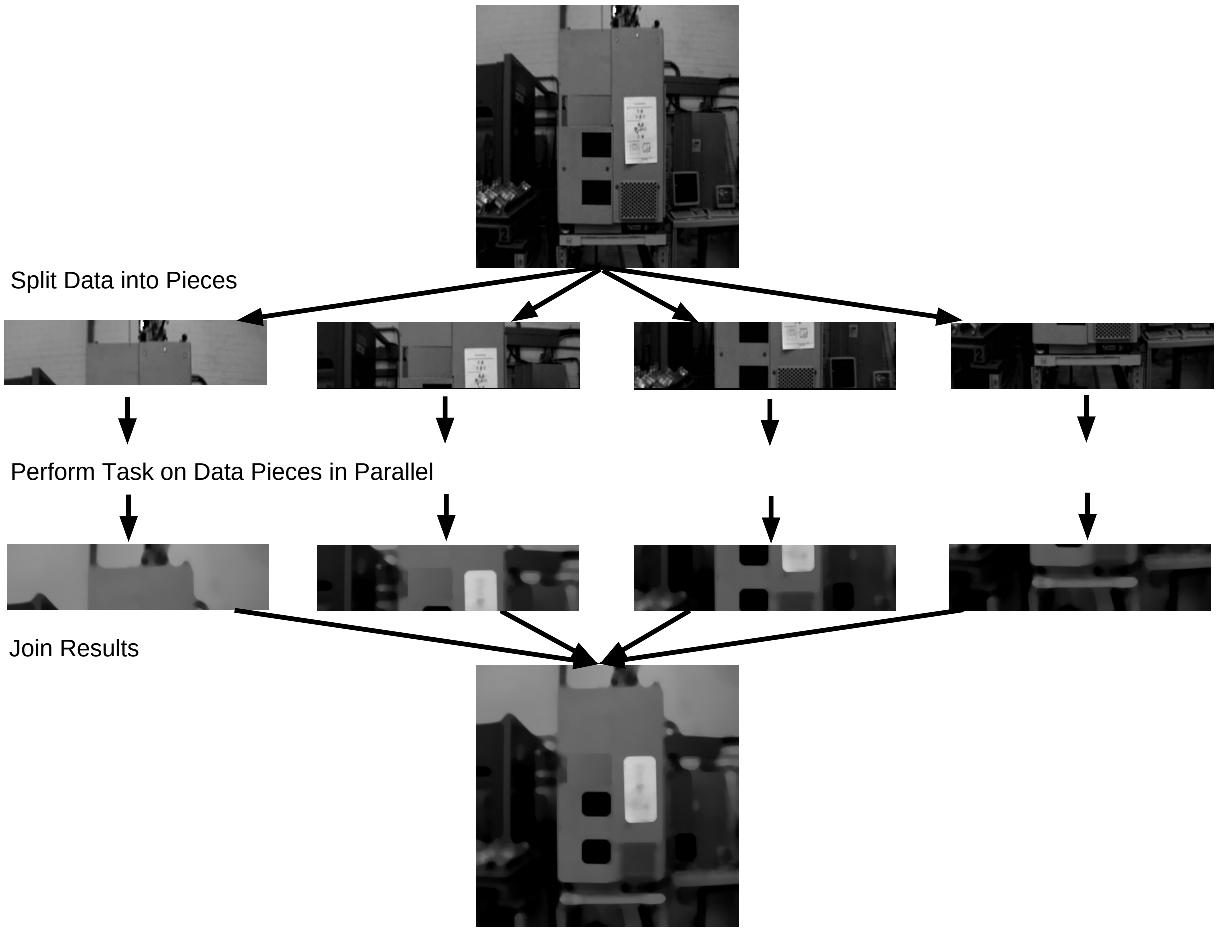


Figure 2.1: Minimize response time using data parallelization. Here, a median filter is applied to different parts of the image simultaneously.

To use AOP, the system parameter *'parallelize_operators'* must be set to *'true'* (which is the default). The number of threads to be used can be set explicitly by setting the system parameter *'thread_num'* to a value between 1 and the available number of cores. By default, all available cores are used for AOP.

```
set_system ('parallelize_operators', 'true')
median_image (Image, ImageMedianAop, 'circle', Radius, 'mirrored')
```

The following code shows an implementation of data parallelization by simulating AOP. As AOP should be simulated, the AOP is turned off first. Then, the image is split into pieces of approximately the same size. This is done by reducing the domains of the images to be handled by the individual threads to a respective sub-region of the domain of the input image. With the qualifier `par_start` (see also the HDevelopUser's Guide, section 8.11.1 on page 283), the individual threads that run in parallel are started. In each thread, the median is calculated for the respective image part. The results of the different threads are collected using vector variables (see also the HDevelopUser's Guide, section 8.6 on page 270).

```
get_system ('processor_num', Cores)
MaxNumberOfSubthreads := 20
MaxThreads := min([Cores,MaxNumberOfSubthreads])
set_system ('parallelize_operators', 'false')
for Threads := 1 to MaxThreads by 1
        get_region_runs (Image, Row, ColumnBegin, ColumnEnd)
        RunsPerThread := |Row| / Threads
        * Split up the image and process each region by a separate thread:
        for Thread := 0 to Threads - 1 by 1
            IndexStart := Thread * RunsPerThread
            if (Thread < Threads - 1)
                IndexEnd := IndexStart + RunsPerThread - 1
            else
                IndexEnd := |Row| - 1
            endif
            gen_region_runs (Region, Row[IndexStart:IndexEnd], \
                             ColumnBegin[IndexStart:IndexEnd], \
                             ColumnEnd[IndexStart:IndexEnd])
            reduce_domain (Image, Region, ImageReduced)
            par_start<ThreadID.at(Thread)> : median_image (ImageReduced, \
                                  ImageMedianThread.at(Thread), 'circle', \
                                  Radius, 'mirrored')
        endfor
        convert_vector_to_tuple (ThreadID, ThreadIDs)
endfor
```

To wait until all involved threads are finished, the operator par_join is called. Then, the results of the individual threads are copied into one image.

```
par_join (ThreadIDs)
* Free references to thread IDs.
ThreadID.clear()
ThreadIDs := []
* Merge results
full_domain (ImageMedianThread.at(0), ImageMedianPar)
for ImagePart := 1 to Threads - 1 by 1
    overpaint_gray (ImageMedianPar, ImageMedianThread.at(ImagePart))
endfor
```

The runtimes of the different cases are visualized in figure 2.2 for a machine with 4 cores. The manual parallelization has no advantage compared to the automatic operator parallelization. Due to some overhead caused by HDevelop, the runtime of the simulated AOP processing is even slightly larger than that of the AOP. But both are significantly faster than the sequential processing.

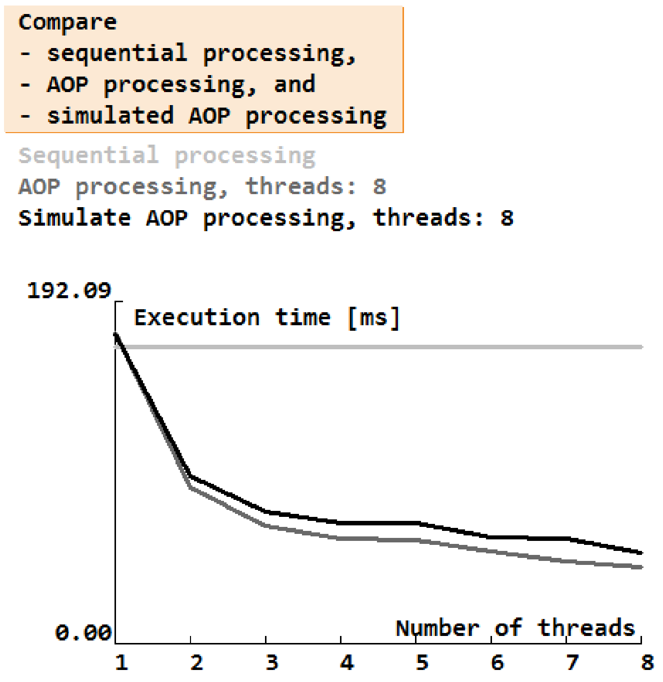


Figure 2.2: Runtimes for applying the median filter without AOP, with AOP, and with parallel programming simulating AOP on a machine with 4 cores / 8 Hardware Threads.

2.1.2 Task Parallelization

Task parallelization means that, e.g., on the same image, different tasks are performed in different threads (see figure 2.3).

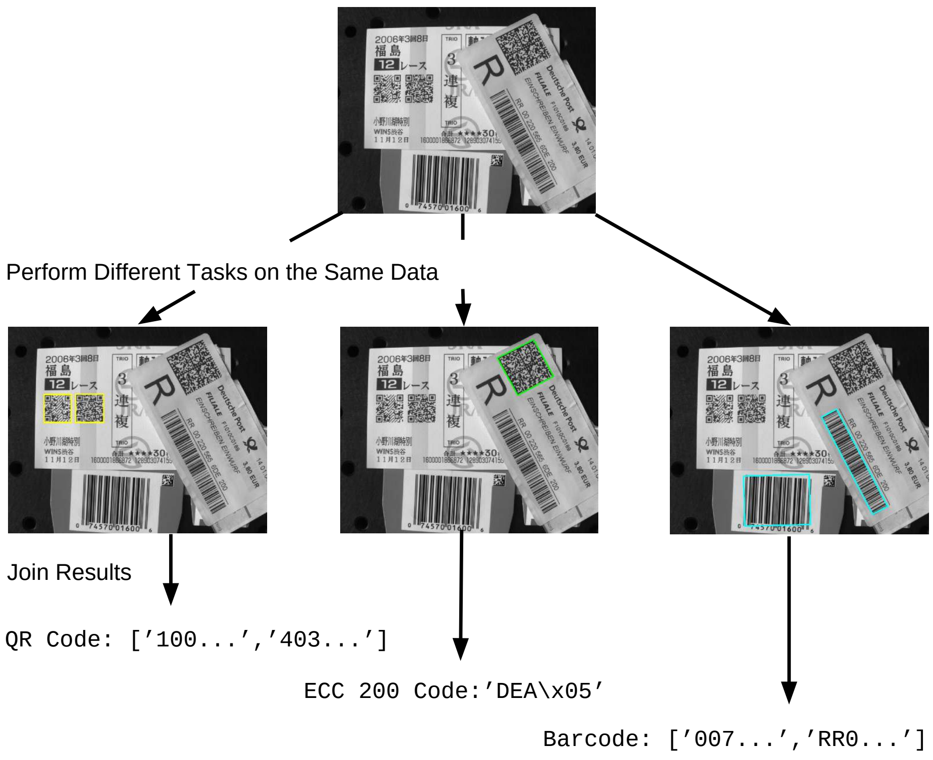

Perform Different Tasks on the Same Data
Join Results
QR Code: ['100...','403...']
ECC 200 Code:'DEA\x05'
Barcode: ['007...','RR0...']

Figure 2.3: Minimize response time using task parallelization. On the same image, three tasks are processed in parallel: QR code reading, ECC 200 reading, and bar code reading.

The HDevelop example program `hdevelop/Control/par_start.hdev` reads bar codes and different types of data codes in parallel.

After the creation and training of the bar code and data code models, for each reading task a corresponding procedure is called in a separate thread using the `par_start` qualifier: a procedure to read bar codes, a procedure to read ECC 200 codes, and a procedure to read QR codes. The operator `par_join` waits for all sub-threads to finish before the results of the individual threads are used for further processing (e.g., the display of results).

```
par_start<ThreadQRCode> : find_qr_codes (Image, QrCodeXlds, QrCodeModel, \
                                          QrCodeHandlesSeq, \
                                          QrCodeStringsSeq)
par_start<ThreadECC200> : find_ecc200_codes (Image, Ecc200Xlds, Ecc200Model, \
                                              Ecc200Handles, Ecc200Strings)
par_start<ThreadBarcode> : find_bar_codes (Image, BarCodeRegions, \
                                            BarCodeModel, BarCodeStrings)
par_join ([ThreadQRCode,ThreadECC200,ThreadBarcode])
dev_display (QrCodeXlds)
dev_display (Ecc200Xlds)
dev_display (BarCodeRegions)
```

Note that the example program not only shows how to parallelize the three identification tasks, but also provides hints for which kinds of tasks parallelization is reasonable. Additionally, the speedup of the parallelization is demonstrated by a comparison of the runtimes needed for the parallelized procedure calls and the corresponding sequentially applied procedure calls.

2.2 Maximize Throughput

Especially when dealing with streaming data, the throughput of a system is often more important than its response time.

One way to increase the throughput of a system is pipelining.

2.2.1 Pipelining

Pipelining is frequently associated with the processing of streaming data, like images taken from a camera that monitors objects on a conveyor belt (compare figure 2.4). In this case, the image of object 3 may be acquired at the same time as the image of object 2 is processed and at the same time as the image processing results of object 1 are evaluated.

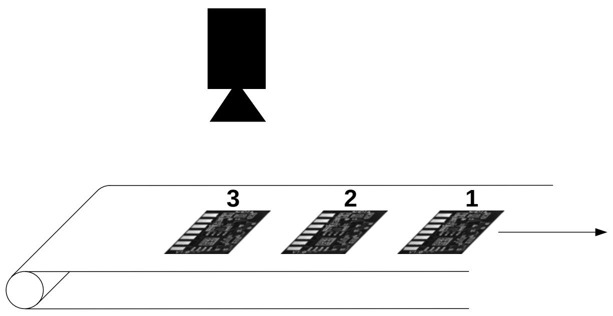


Figure 2.4: Setup that might produce streaming data.

2.2.1.1 Basic Design of a Pipeline

The individual processing steps, e.g., image acquisition, image processing, and result evaluation, are executed by different stages that are connected by message queues such that the output of one stage is the input of the next one (see figure 2.5). Thus, a pipeline is composed of (typically) multiple producer/consumer pairs.

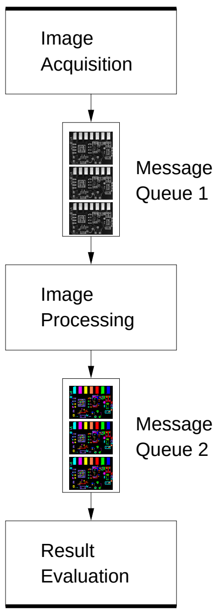


Figure 2.5: Simple pipeline that consists of the three stages image acquisition, image processing, and result evaluation.

The pipeline in figure 2.5 consists of three stages, which are connected by two message queues. All three stages run in separate threads that run in parallel. The image acquisition stage acquires the images. Each image is put into a message, which then is enqueued into the message queue 1. The message queues act as FIFO (first in, first out) buffers, which connect the consecutive stages. If the image processing stage is ready to process the next image and as soon as there is a message available in message queue 1, the image processing stage dequeues a message from the head of message queue 1. It processes the image that is contained in the message and adds the processing results to the message. Then it enqueues this message to message queue 2. Finally, the result evaluation stage dequeues the message from the head of message queue 2 and evaluates the results, e.g., by initiating some action like blowing-out defective products.

As the threads run in parallel, the result evaluation for the first image may be executed parallel to the processing of the second image and parallel to the acquisition of the third image. Typically, the processing time of the different stages is unequal. For example, in figure 2.6, the image processing stage takes twice as long as the result evaluation stage.

The cycle time of a pipeline is the time between the completion of consecutive tasks. The throughput of a pipeline is the inverse of its cycle time. Note that the throughput of a pipeline is limited by the throughput of its slowest stage.

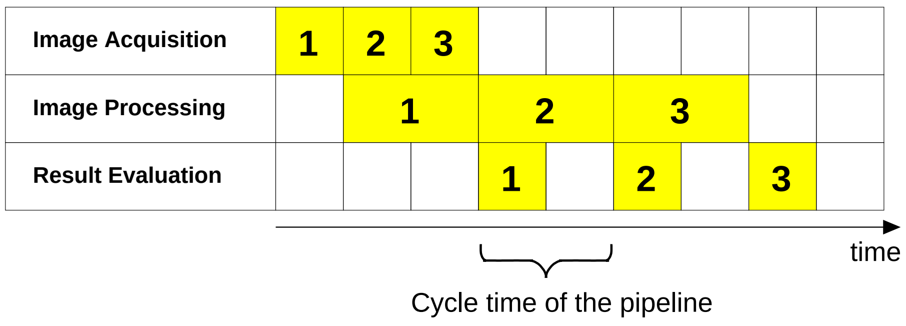


Figure 2.6: Scheduling diagram for a simple pipeline.

Because the individual stages of a pipeline are executed in parallel, its throughput is typically higher than the throughput of a program that executes the stages sequentially (see figure 2.7).

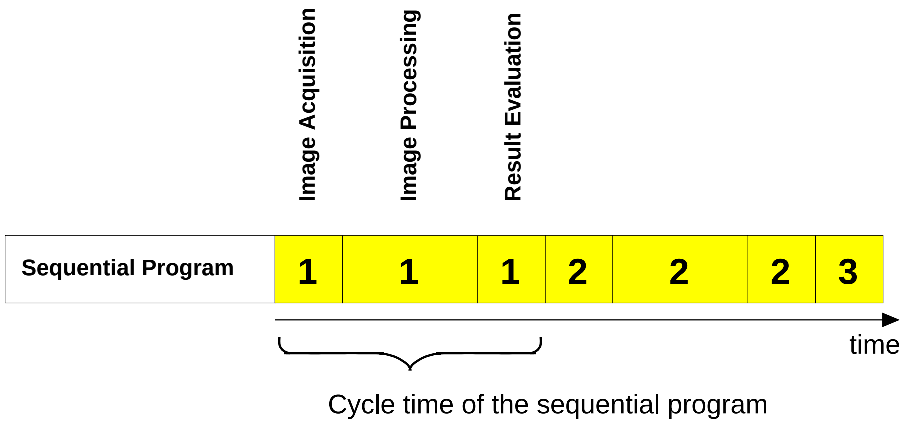


Figure 2.7: Scheduling diagram for a sequential program.

Example Implementation of the Pipeline

The implementation of a pipeline follows the pattern:

1. Put the functionality of the stages into individual procedures.
2. Create and configure message queues.
3. Prepare data structures for a clean termination of the pipeline.

4. Start the procedures in individual threads.

The HDevelop example program `hdevelop/System/Multithreading/pipeline_one_thread_per_stage.hdev` shows an implementation of a pipeline similar to the one referred to in figure 2.5 on page 13 and figure 2.6.

First, the functionality of the different stages has to be put into individual procedures. Here, the three procedures `acquire_images`, `process_images`, and `evaluate_results` are used. The data that must be passed from one stage to the next stage will be passed through message queues. As we have three stages, we need two message queues to connect theses stages. They are created with the operators `create_message_queue`:

```
create_message_queue (QueueOriginalImage)
create_message_queue (QueueImageProcessingResult)
```

Typically, the size of the message queues should be limited to control the memory usage of the program. This is done by the following calls:

```
MessageQueueMaxMessageNum := 200
set_message_queue_param (QueueOriginalImage, 'max_message_num', \
                         MessageQueueMaxMessageNum)
set_message_queue_param (QueueImageProcessingResult, 'max_message_num', \
                         MessageQueueMaxMessageNum)
```

Note that the size of the message queues should be chosen large enough to allow them to handle short-term peak loads. If, e.g., for a short period of time, the images are acquired faster than they can be processed, they will be buffered in the message queue.

To be able to stop the image acquisition, we use an event that is monitored inside the procedure `acquire_images`.

```
create_event ('', '', StopAcq)
```

Furthermore, we propagate the information that the image acquisition has been stopped with a respective message. This message is created by the image acquisition stage as soon as the image acquisition is stopped. It indicates that no further messages will follow. To identify this message, we define a particular message key:

```
TerminationKey := 'terminate'
```

Then, we can start the procedures in individual threads:

```
par_start<AcquisitionThread> : acquire_images (MeanAcquisitionInterval, \
                                               StopAcq, TerminationKey, \
                                               QueueOriginalImage)
par_start<ImageProcessingThread> : process_images (QueueOriginalImage, \
                                                   NumAOPThreadsForImageProcessing, \
                                                   TerminationKey, \
                                                   QueueImageProcessingResult)
par_start<ResultEvaluationThread> : evaluate_results (QueueImageProcessingResult, \
                                                      TerminationKey, \
                                                      WindowHandle)
```

In the example program, the pipeline runs just a few seconds and is then terminated by signaling the event `StopAcq`:

```
signal_event (StopAcq)
```

Now, the image acquisition stage, which monitors the state of the event `StopAcq`, terminates and sends a message to the subsequent stages, which will terminate themselves as soon as they evaluate this message.

Finally, we must wait until all threads are finished.

```
par_join ([AcquisitionThread,ImageProcessingThread,ResultEvaluationThread])
```

Example Implementation of the Image Acquisition Stage

The implementation of the first stage (here represented by the procedure `acquire_images`) follows the pattern:

1. Acquire data.
2. Create new message.
3. Add data and meta information to the message.
4. Enqueue the message to the output message queue.
5. Check and handle the event that signals the termination of the stage.

The image acquisition stage acquires images in a loop, which is not exited until either the image acquisition is terminated by signaling the termination event or the message queue that connects the image acquisition stage with the image processing stage reaches its limit.

In our example, the image acquisition is simulated by picking a random image from an array of pre-read images and waiting some time to simulate a realistic time interval between the acquisition of two images. In real applications, these two lines must be replaced to acquire real images.

```
repeat
    wait_to_simulate_non_trivial_acquisition_time (MeanAcquisitionInterval, \
                                                   ImageId, \
                                                   LastAcquisitionTime, \
                                                   LastAcquisitionTime)
    acquire_random_image (Images, Image)
```

For each image, we create a message and store the image together with an image ID and a time stamp in this message.

```
    create_message (MessageImg)
    set_message_obj (Image, MessageImg, 'image')
    set_message_tuple (MessageImg, 'image_id', ImageId)
    set_message_tuple (MessageImg, 'acquisition_time', AcquisitionTime)
```

Finally, the message is enqueued into the message queue that connects the image acquisition stage with the image processing stage. Because the size of our message queues is limited, we must take care of the case that the message queue already contains its maximally allowed number of messages. For this, we use a procedure that can be configured to handle this case in the desired way.

```
    enqueue_message_to_limited_queue (MessageImg, QueueOut, \
                                      'throw_exception')
```

Here, we want to throw an exception if the message queue is at its limit. In real applications, this case must be handled individually and with care. It indicates that one of the following stages cannot handle the images in the required speed. Because the reduction of the image acquisition rate is most often not the desired solution, the speed of the pipeline must be increased. See section 2.2.1.2 on page 18 for possible ways to speedup the pipeline.

Finally, we check if the image acquisition should be terminated by checking the state of the event `StopAcq`. As long as this event is not signaled, the image acquisition stage continues to acquire images.

```
    try_wait_event (StopAcq, Busy)
until (not Busy)
```

If the event `StopAcq` is signaled, the image acquisition loop will be left and the complete pipeline can be terminated. To do this in a way that all acquired images are still being processed, we create a message that contains the above defined termination key and enqueue it as the final message to the message queue. In this case, we must ensure that the message will be enqueued to the message queue. Therefore, we do not want to throw an exception if the message queue is at its limit but rather wait until the message can be enqueued.

```
create_message (TerminationMessage)
set_message_tuple (TerminationMessage, TerminationKey, 'true')
enqueue_message_to_limited_queue (TerminationMessage, QueueOut, 'wait')
```

Example Implementation of the Image Processing Stage

The implementation of intermediate stages (here represented by the procedure `process_images`) follows the pattern:

1. Dequeue a message from the input message queue.
2. Process the data contained in the message.
3. Add processing results to the message.
4. Enqueue the message to the output message queue.

All these steps are performed inside a loop, which exits if the pipeline is terminated.

First, a message is dequeued from the input message queue:

```
while (1)
    dequeue_message (QueueIn, [], [], MessageHandle)
```

Then, we check if the image acquisition was terminated and thus we have to terminate the pipeline:

```
    get_message_param (MessageHandle, 'key_exists', TerminationKey, \
                       Terminate)
```

If we have to terminate the pipeline, the following two steps are needed:

1. Enqueue the termination message to the output queue to inform subsequent stages about the termination.
2. Exit the loop.

```
    if (Terminate)
        enqueue_message_to_limited_queue (MessageHandle, QueueOut, 'wait')
        break
    endif
```

If no termination was requested, we can extract the input data (here: image) from the message, perform some action (here: image processing), and add the result (here: number of defects) to the message:

```
    get_message_obj (Image, MessageHandle, 'image')
    image_processing (Image, ImageReduced, RegionUnion, RingSize, \
                      PolarResolution, SmoothX, Number)
    set_message_tuple (MessageHandle, 'num_defects', Number)
```

Finally, we enqueue the message to the output queue:

```
    enqueue_message_to_limited_queue (MessageHandle, QueueOut, 'wait')
endwhile
```

Here it is not necessary to throw an exception if the output queue reached its limit. We just wait until we can enqueue the message. As long as we are waiting, we cannot dequeue further messages from the input queue. If this waiting state takes too long, the input queue will reach its limit and a respective exception will be thrown by the image acquisition stage.

Example Implementation of the Result Evaluation Stage

The implementation of the final stage (here represented by the procedure `evaluate_results`) follows the pattern:

1. Dequeue a message from the input message queue.
2. Process the data contained in the message.

All these steps are performed inside a loop, which is not exited until the pipeline is terminated.

First, a message is dequeued from the input message queue:

```
while (1)
    dequeue_message (QueueIn, [], [], MessageHandle)
```

Then, we check if the image acquisition was terminated and thus we have to terminate the pipeline:

```
    get_message_param (MessageHandle, 'key_exists', TerminationKey, \
                       Terminate)
    if (Terminate)
        break
    endif
endwhile
```

Now, we can evaluate the results. In the example, we just calculate response time, processing time, and the mean acquisition interval and visualize the data. In real applications, the results can, e.g., be used to control the elimination of defective parts.

Runtime Behavior

Figure 2.8 shows the runtime behavior of the simple pipeline realized in the above described HDevelop example program. On the horizontal axis, the last 720 images are plotted. The orange line shows the acquisition interval, i.e., the time between the acquisition of two consecutive images. The acquisition interval shows some variations, which are specifically caused by the image acquisition procedure. The magenta line shows the time needed for the image processing stage, which is fairly constant over the images. The blue line shows the response time, i.e., the time between the image acquisition and the completion of the result evaluation.

In figure 2.8 (a), the images are acquired in approximately equal intervals of 6 ms. Note that the image acquisition must not be faster than the slowest of the following stages of the pipeline. In our example, this stage is the image processing stage, which takes about 4.5 ms. With the above mentioned image acquisition interval of 6 ms, we can achieve a throughput of $^1/_{(6\,ms)} \approx 160$ images/sec with a nearly constant response time.

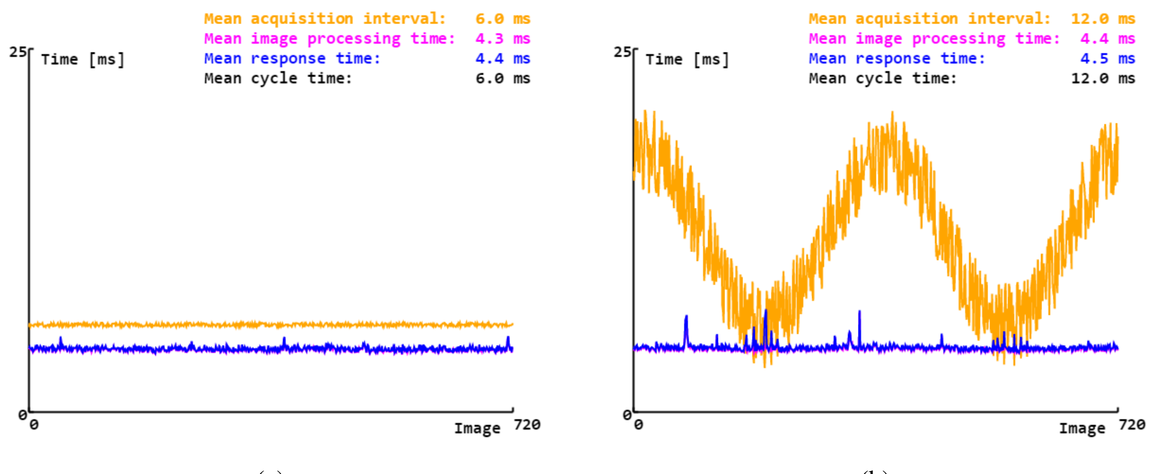
In figure 2.8 (b) and (c), the image acquisition interval varies over time leading to short-term peak loads. As long as the pipeline (more precisely: the slowest stage following the image acquisition stage) is fast enough, this has no negative effect on the response time. This can be seen in figure 2.8 (b), where the images are always acquired slower than they are processed. Naturally, this limits the achievable throughput, because at any time, the image acquisition speed may not exceed the speed of the following stages. In this configuration of our example, only $^1/_{(12\,ms)} \approx 80$ images/sec can be processed, again with a nearly constant response time.

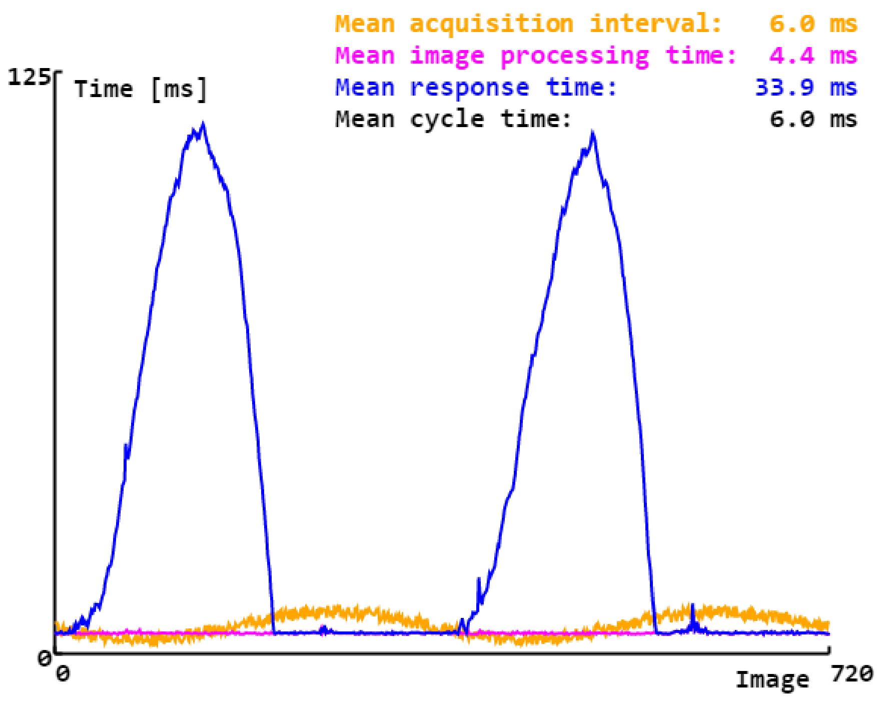
If the response time is less important, for a limited period of time, the image acquisition interval can even drop below the time required to process one image (see figure 2.8 (c)). As long as the image processing is faster than the image acquisition (magenta line is below the orange line), the response time lies only marginally above the image processing time. In this case, the message queues contain typically at most one message. As soon as the acquisition interval is shorter than the time required to process one image, the acquired images must be buffered in the input message queue of the image processing stage. The response time for those images, which must wait in the message queue for being processed, increases dramatically. As soon as the image acquisition is again slower than the image processing, the number of buffered images decreases and with this also the response time for newly acquired images. Note that the message queues must be configured large enough to be able to buffer the images that are waiting for being processed. In this configuration of our example, again $^1/_{(6\,ms)} \approx 160$ images/sec can be processed, but with a response time that may be dramatically slower.

2.2.1.2 Speedup the Pipeline

There are two possible ways to increase the speed of a pipeline:

1. Split the slowest stage into multiple stages.
2. Execute the slowest stage in parallel on multiple threads.





(c)

Figure 2.8: Runtime behavior of the simple pipeline: (a) Approximately constant acquisition interval. (b) Varying acquisition interval. (c) Varying acquisition interval leading to short-term peak loads..

Both approaches try to reduce the mean processing time required for the slowest stage of the pipeline, as this limits the speed of the whole pipeline.

While splitting the slowest stage into multiple stages is straightforward and probably the simpler approach, its effectiveness depends on the possibility to split the stage into pieces of approximately equal size (run time). If this is not possible, the speed of the individual stages again vary and the speed of the pipeline is suboptimal.

Figure 2.9 shows the scheduling diagram of a pipeline where the image processing stage was split into two stages. Here, it was not possible to split the image processing stage into two stages of equal size. Instead, the first part is three times larger than the second part. Compared to the simple pipeline described in figure 2.6 on page 14, the cycle time of the pipeline was only slightly reduced. The stage 'image processing 1' is still significantly slower than the other stages and thus limits the throughput of the pipeline.

The second approach, the parallel execution of the slowest stage on multiple threads, requires a little more implementation effort, but typically, it is easier to reach a more similar workload of the threads of this stage. Figure 2.10 shows such a pipeline, where the image processing stage runs on two threads in parallel.

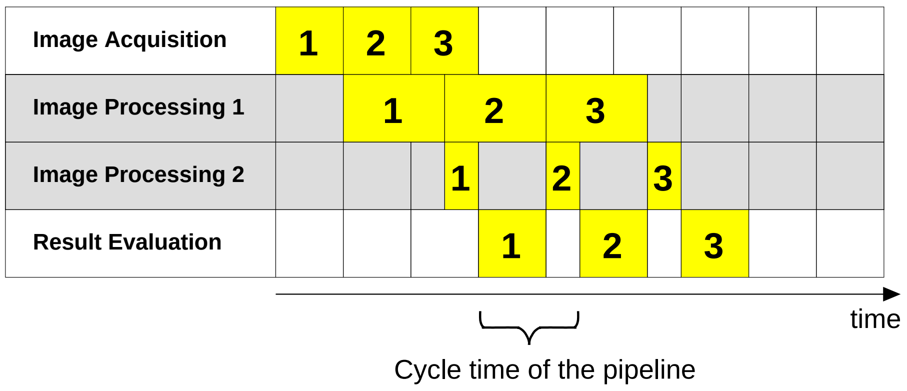


Figure 2.9: Scheduling diagram for a pipeline with two image processing stages of unequal size.

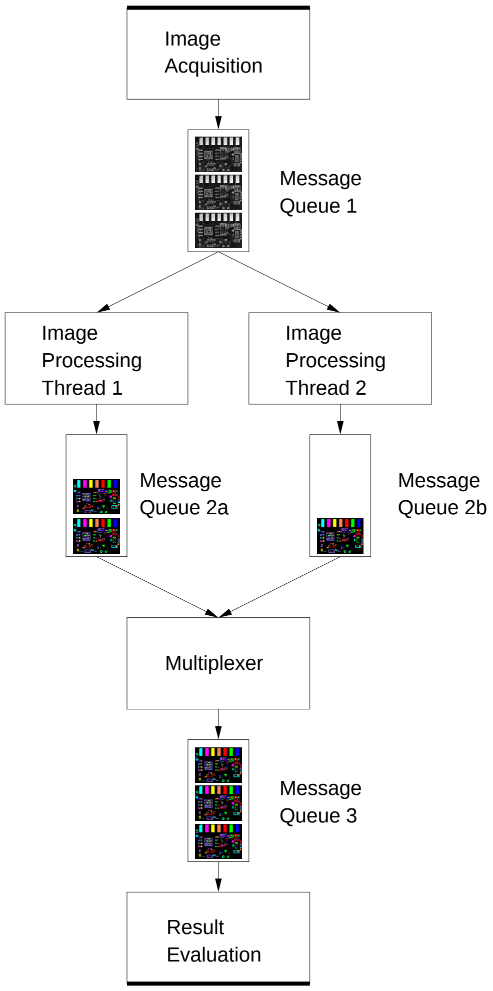


Figure 2.10: Pipeline where the image processing stage runs on two threads in parallel. Note the multiplexer, which ensures that the results appear in the correct order.

Depending on the image contents, the image processing task that was started last may finish before the task that was started first.

If it is required that the results appear in the same order the tasks were started, a multiplexer must be inserted after the parallel stage into the pipeline. A multiplexer guarantees that the tasks are returned in the same order they were started. It combines the output message queues of the parallel stage into one single message queue that contains the tasks in correct order.

If the order of the results is irrelevant, e.g., if the results are written into a data base, no multiplexer is required. In those cases, the images can still be identified by their image ID.

Figure 2.11 shows the scheduling diagram for the above discussed pipeline. The image processing of the second image starts while the first image is still being processed. Compared to figure 2.6 on page 14, the parallelization of the image processing stage (nearly) doubles the throughput of the pipeline.

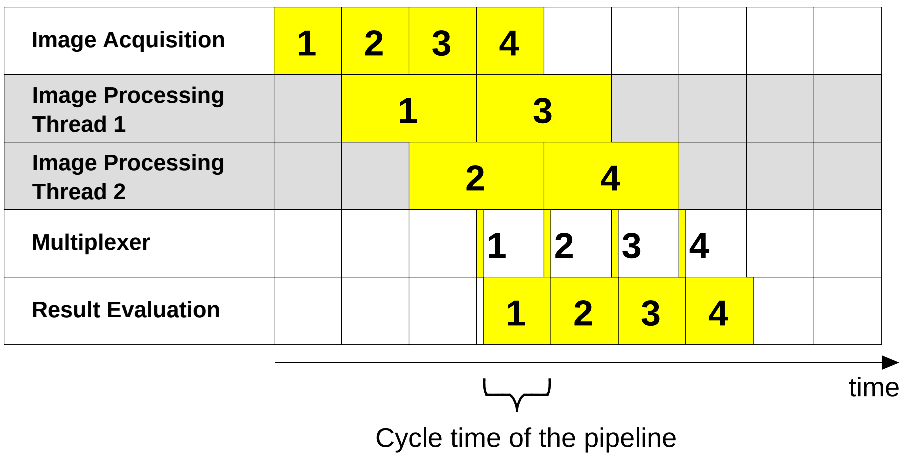


Image Acquisition
1
2
3
4
Image Processing Thread 1
1
3
Image Processing Thread 2
2
4
Multiplexer
1
2
3
4
Result Evaluation
1
2
3
4
time
Cycle time of the pipeline

Figure 2.11: Scheduling diagram for a pipeline where the image processing stage runs on two threads in parallel.

Example Implementation of the Pipeline

The HDevelop example program `hdevelop/System/Multithreading/pipeline_multiple_threads_per_stage.hdev` shows an implementation of a pipeline where the image processing stage runs on multiple threads in parallel, i.e., a pipeline which is similar to the one referred to in figure 2.10 on page 20 and figure 2.11.

In the following, only the changes are described that are necessary to convert the simple pipeline as described in section 2.2.1.1 on page 13 into a pipeline that runs the image processing stage on multiple threads.

In addition to the message queue that delivers the image processing results to the result evaluation stage, we need multiple message queues that connect the image processing stage with the multiplexer:

```
for I := 0 to NumImageProcessingThreads - 1 by 1
    create_message_queue (QueueImageProcessingResultsVector.at(I))
    set_message_queue_param (QueueImageProcessingResultsVector.at(I), \
                             'max_message_num', MessageQueueMaxMessageNum)
endfor
convert_vector_to_tuple (QueueImageProcessingResultsVector, \
                         QueuesImageProcessingResult)
```

Instead of starting only one thread for the image processing stage, we must start multiple image processing threads as well as one multiplexer thread:

```
for I := 0 to NumImageProcessingThreads - 1 by 1
    par_start<ImageProcessingThreadVector.at(I)> : process_images (QueueOriginalImage, \
                                            NumAOPThreadsForImageProcessing, \
                                            TerminationKey, \
                                            QueueImageProcessingResultsVector.at(I))
endfor
convert_vector_to_tuple (ImageProcessingThreadVector, \
                         ImageProcessingThreads)
par_start<MultiplexerThread> : multiplexer (QueuesImageProcessingResult, \
                                            'image_id', TerminationKey, \
                                            QueueImageProcessingResult)
```

We must also wait for the additional threads to finish:

```
par_join ([AcquisitionThread,ImageProcessingThreads,MultiplexerThread, \
          ResultEvaluationThread])
```

In order to ensure a proper termination of the pipeline, we must slightly modify the respective part of the procedure `process_images`. In addition to enqueuing the termination message only to the output queue, we must enqueue this message also to the input queue to inform the other threads that are running the procedure `process_images`, as well:

```
    get_message_param (MessageHandle, 'key_exists', TerminationKey, \
                       Terminate)
    if (Terminate)
        enqueue_message_to_limited_queue (MessageHandle, QueueIn, 'wait')
        enqueue_message_to_limited_queue (MessageHandle, QueueOut, 'wait')
        break
    endif
```

Finally, we must implement the multiplexer. The multiplexer ensures that the result evaluation stage receives the processing results in the correct sequential order. Note that the messages do not need to be sorted or reordered. Because of the fact that each input message queue is locally ordered, the multiplexer only has to look for the next message in the sequence by simultaneously monitoring the heads of all input message queues. It determines the required order of the messages from the image ID, which was set in an increasing order by the image acquisition stage.

The basic functionality of the multiplexer is provided by the following lines of code:

```
NumInputQueues := |QueuesIn|
LastId := -1
Buffers := gen_tuple_const(NumInputQueues,-1)
Ids := gen_tuple_const(NumInputQueues,LastId)
while (1)
    for I := 0 to NumInputQueues - 1 by 1
            if (Buffers[I] == -1)
                dequeue_message (QueuesIn[I], [], [], MessageHandle)
                get_message_tuple (MessageHandle, SortIdKey, Id)
                Buffers[I] := MessageHandle
                Ids[I] := Id
            endif
            if (Buffers[I] != -1 and Ids[I] == LastId + 1)
                enqueue_message_to_limited_queue (Buffers[I], QueueOut, \
                                                  'wait')
                Buffers[I] := -1
                LastId := Ids[I]
            endif
    endfor
endwhile
```

The remainder of the multiplexer's code, which can be found in the HDevelop example program `hdevelop/System/Multithreading/pipeline_multiple_threads_per_stage.hdev`, mainly deals with its termination if the pipeline is terminated and with the — unlikely — case that some messages are missing and therefore, there are gaps in the sequence of the image IDs.

Runtime Behavior

The runtime behavior of the pipeline where the image processing stage runs on two threads is shown in figure 2.12. It looks similar to that of the simple pipeline with only one thread per stage (see figure 2.8 on page 19), but with the important difference that the acquisition interval may be significantly shorter. Thus, the throughput of the system was significantly increased.

2.3 Enhance Responsiveness

Responsiveness can be enhanced on the one hand using a producer consumer model (section 2.3.1), which is no parallelization concept but a method that can be used within any parallelization concept, and on the other hand by

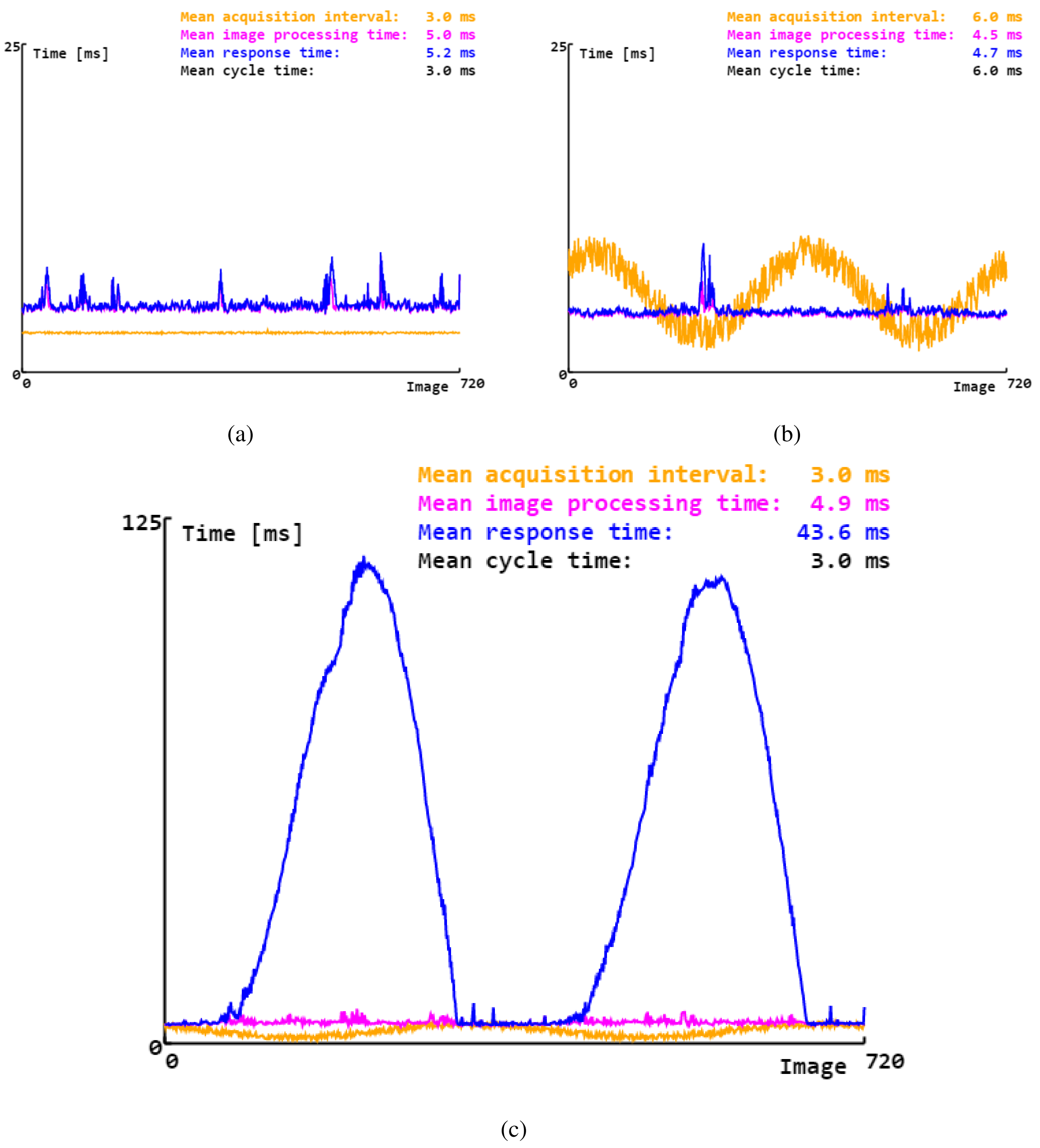


Figure 2.12: Runtime behavior of a pipeline where the image processing stage runs on two threads: (a) Approximately constant acquisition interval. (b) Varying acquisition interval. (c) Varying acquisition interval leading to short-term peak loads..

specific parallelization scenarios (section 2.3.2 on page 24).

2.3.1 Producer Consumer Model

The producer consumer model is not a parallelization concept or scenario but a method that can be used within all different parallelization approaches. In general, this model describes that one thread, the producer, provides some data which one or several threads, the consumer(s), use(s) for further processing. A thread can be both producer and consumer, e.g., within a pipeline, which consists of several producer consumer processes in row. The communication between the producer and consumer threads is realized by message queues.

Note that the processing times are not always the same. Thus, if, e.g., a producer thread acquires images and the consumer threads process them, images can get lost if the processing threads are not ready to catch them in time. Images have to be buffered and special care has to be taken to avoid a buffer overflow. How to do that, e.g., for a pipeline, is demonstrated in section 2.2.1 on page 13.

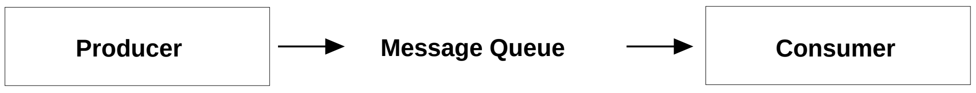


Figure 2.13: Enhance the responsiveness using a producer consumer model.

2.3.2 Further Parallelization Scenarios

Different parallelization settings exist that enhance the responsiveness. In figure 2.14, e.g., a setup is shown in which moving objects are viewed from different angles at different subsequent times. Each image is processed in a separate thread. The challenge is to synchronize the results after the processing.

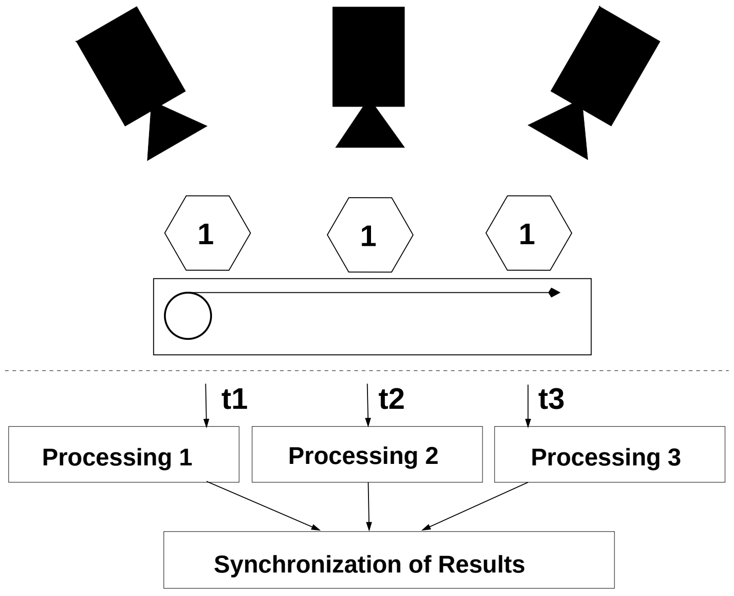


Figure 2.14: Synchronization of results needed after processing images of the same object at different times (t).

Other examples comprise

- scenarios in which input actions via keys, mouse clicks, IA, I/O etc. are expected or
- scenarios in which a synchronization between the GUI and the processing steps is needed (which is not relevant in HDevelop, but when parallelizing within the language interfaces).

In any case, synchronization is often tricky. Thus, the need for synchronization should always be reduced as much as possible.

Chapter 3

Where to Find More Information?

Information related to parallelization can be found at different parts of the HALCON documentation. In the following, we provide you with hints, where to look for further information.

General Information

General information related to parallelization can be found in the Programmer's Guide, chapter 2 on page 15. In particular, for the automatic operator parallelization, it is described how to initialize HALCON, which levels of data parallelism are available for the different operators and how to query them for specific operators. For parallel programming using HALCON, amongst others the different levels of reentrancy are explained, program design issues are addressed, and a list of suitable examples using the language interfaces is provided. For threading issues with HALCON graphic operators, the restrictions for the different platforms are listed. Finally, it is explained how to customize the parallelization mechanisms and what to consider when using image acquisition interfaces on multi-core or multi-processor hardware.

Note that HALCON is designed for *shared-memory systems*, i.e., systems in which multiple processors share a common memory as it is the case for typical multi-processor or multi-core boards. The main reason is that only in a shared-memory system threads can share the HALCON object database and do not need to copy images. This limitation means that HALCON's parallelization techniques are not suited to the use on workstation clusters or other multi-processor or multi-core hardware that does not offer shared memory.

When Using HDevelop

If you use HDevelop, you should have a look at the HDevelopUser's Guide. In particular, the section 8.11 on page 283 addresses issues that are specific for parallelization within HDevelop. Amongst others, the qualifier `par_start` and the HDevelop operator `par_join` are explained in more detail, the execution of procedure and operator calls within HDevelop is addressed, and it is described how to inspect, suspend, and resume threads within HDevelop.

Additionally, the HDevelopUser's Guide, section 10.2.3.3 on page 313 contains information about restrictions concerning the export of manually parallelized code.

When Using the HDevEngine

If you use the HDevEngine, additional information can be found in the Programmer's Guide, chapter 21 on page 137, section 22.2.5 on page 150 (C++), and section 23.2.7 on page 164 (.NET).

Operator References Related to Parallelization

Several operators exist that are related to parallelization. In particular, the following chapters of the reference manual are important:

- `System -> Parallelization`: These operators configure the AOP.
- `System -> Multithreading`: These operators are relevant for parallel programming, in particular for the communication and synchronization between threads of multi-threaded programs.

- `System -> Parameters`: The operators `get_system` and `set_system` are used to query and configure general settings related to parallelization.
- `Control`: The HDevelop operator `par_join` is used for parallel programming within HDevelop.

The explanation of the slots “Execution Information” (“parallelization”) and “postprocessing” within the operator descriptions in the reference manual can be found in the Extension Package Programmer’s Manual, section 2.2.13 on page 24 and section 2.3.7 on page 31.

Examples

A list of suitable examples for parallel programming using the language interfaces is provided in the Programmer’s Guide, section 2.2.4 on page 18.

For HDevelop example programs, we recommend to use the `Browse HDevelop Example Programs...` dialog to search for suitable examples.

Chapter 4

Glossary

Thread
A thread is a procedure that runs independently from and in parallel with its main program. It shares global variables and resources with other threads and holds local ones exclusively.

Thread Safe
A procedure is considered to be thread safe when it accesses or manipulates shared data and resources in a manner that guarantees predictable and reproducible results in an arbitrary multithreaded environment.

Reentrant
The term reentrant is sometimes used to mean “efficiently thread-safe”. In HALCON, not all operators are fully reentrant. The different levels of reentrancy are described in the Programmer’s Guide, section 2.2.1 on page 16.

Exclusive
Data, resources or code is accessed or executed exclusively, if it is guaranteed that it can only be accessed or executed by one thread at the same time.

Message Queue
A message queue is a thread safe FIFO (“First In First Out”) list to pass data between threads.

Producer Consumer Model
Within the producer consumer model, some threads (i.e., producers) provide other threads (i.e., consumers) with data.

Data Parallelization
Data parallelization describes a parallelization concept that separates data into independent data groups and processes these data groups simultaneously.

Task Parallelization
Task parallelization describes a parallelization concept that divides the problem into several independent subtasks and executes these subtasks simultaneously.

AOP (automatic operator parallelization)
AOP is a type of data parallelization that is by default used internally for selected/distinguished operators.