Employing MPI collectives for timing analysis on embedded multi-cores

Martin Frieb, Alexander Stegmeier, Jörg Mische, Theo Ungerer

Angaben zur Veröffentlichung / Publication details:

Frieb, Martin, Alexander Stegmeier, Jörg Mische, and Theo Ungerer. 2016. "Employing MPI collectives for timing analysis on embedded multi-cores." In 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), edited by Martin Schoeberl, 10:1–11. Dagstuhl: Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.WCET.2016.10.

Employing MPI Collectives for Timing Analysis on Embedded Multi-Cores

Martin Frieb[1], Alexander Stegmeier[2], Jörg Mische[3], and Theo Ungerer[4]

1 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
martin.frieb@informatik.uni-augsburg.de
2 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
alexander.stegmeier@informatik.uni-augsburg.de
3 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
mische@informatik.uni-augsburg.de
4 **Systems and Networking, Department of Computer Science, University of Augsburg, Augsburg, Germany**
ungerer@informatik.uni-augsburg.de

Abstract

Static WCET analysis of parallel programs running on shared-memory multicores suffers from high pessimism. Instead, distributed memory platforms which communicate via messages may be one solution for manycore systems. Message Passing Interface (MPI) is a standard for communication on these platforms. We show how its concept of collective operations can be employed for timing analysis. The idea is that the worst-case execution time (WCET) of a parallel program may be estimated by adding the WCET estimates of sequential program parts to the WCET estimates of communication parts. Therefore, we first analyse the two MPI operations MPI_Allreduce and MPI_Sendrecv. Employing these results, we make a timing analysis of the conjugate gradient (CG) benchmark from the NAS parallel benchmark suite.

1998 ACM Subject Classification C.1.2 [Processor Architectures] Multiple-Instruction-Stream, Multiple-Data-Stream Processors (MIMD), C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems, D.1.3 Concurrent Programming, D.2.13 [Reusable Software] Reusable libraries, J.7 [Computers in Other Systems] Real time

Keywords and phrases Real Time, Network on Chip, WCET, Timing Analysis, MPI

Digital Object Identifier 10.4230/OASIcs.WCET.2016.10

1 Introduction

Future embedded real-time systems might realise high performance through high parallelism with many cores. For increasing core numbers, shared memory puts strong limitations on static WCET analysis: it always has to be assumed that all nodes access the memory before the own request is processed by the memory controller (cf. [12, 5]). Furthermore, modern multi- and manycore architectures employ networks on chip (NoCs) to connect cores.

Therefore, parallel platforms with distributed memory and message-based communication might be the way to go (cf. [8]). Message Passing Interface (MPI) [4] is the standard for message-based communication which is widely used in high-performance computing. It encapsulates communication not only in single requests, but also in collective operations,

where all participating cores work together. For example, one (master) core distributes data and coordinates computation which is done by several (slave) cores. This is achieved by executing the same code on all cores and making case distinctions at some points to differentiate between master and slave cores. Moreover, this characteristic is beneficial for timing analysis of a parallel program: worst-case response times (WCRTs) are low and all cores share the same state in the program. Communication steps work as implicit barriers: Cores synchronise and afterwards go on executing sequential code.

However, only few research already took place in the field of real-time message passing parallel programs. Metzlaff et al. [9] described that an overall WCET estimate of a parallel program may be determined by adding the WCET estimate of communication parts to the WCET estimates of sequential program parts. They presumed a predictable NoC, simple cores and distributed memory. Now we go one step further: instead of only determining the worst-case traversal times (WCTTs) for the communication, the WCETs of MPI collectives including the WCTTs are estimated. Then, these collectives' WCET estimates can be added to the WCET estimates of sequential program parts to get an overall WCET estimate.

Since MPI collective operations can be reused by *any* MPI program, the corresponding timing analysis may also be reused. Thus, the analysis effort for MPI collective operations is necessary only once for a given hardware platform and WCTT behaviour (schedule, see Section 2.3). Then, the only remaining work at new programs is to WCET analyse the sequential program parts.

The contributions of our paper are the following: First, we show that a generic timing analysis of MPI collectives is possible for a given hardware platform and generic schedule. We formulate equations with hardware- and schedule specific parameters which can be used to determine the WCET estimate of a MPI operation. Then, we utilise these equations to determine the WCET estimate of a benchmark, where we determined sequential WCET estimates and joined them with WCET estimates of MPI collective operations.

The rest of the paper is structured as follows: In the next Section, we present related work and backgrounds about (real-time) MPI and the setting for our analysis. Afterwards, we analyse MPI collective operations in Section 3 and utilise the results for the analysis of the conjugate gradient (CG) benchmark in Section 4. Finally, we conclude our paper in Section 5 and give an outlook to future work.

2 Related Work and Background

2.1 MPI

MPI is the de-facto standard for message passing [4]. It encapsulates all communication between cores or distributed systems. MPI programs are typically written in a way that all cores execute the same code.

Many MPI programs utilise collective operations to distribute a computation and gather results. These are operations which are executed by all cores of a group[1]. A simple example would be a barrier, but data exchange also often works with collective operations, e. g. MPI_Scatter or MPI_Gather. MPI_Scatter distributes data stored at one core to all participating cores. Afterwards, each core has an equally sized portion of data to work with. MPI_Gather is its counterpart – it collects data from several cores to compound it.

[1] At MPI, groups help to specify which cores communicate together. A group may have any size – only two cores, but also all cores.

The first step towards a timing-analysable MPI was MPI/RT, a standard for real-time MPI [6, 14], an extension of MPI 1.1 from 1998. On the one hand, its extensions are quite broad. On the other hand, MPI/RT is quite old, does not respect NoCs and is not commonly used. In our implementation we focused on being simple and time-predictable and on following the widespread standard MPI [4].

In the context of real-time multi- and manycore architectures, Sørensen et al. [16] already analysed simple MPI operations for the Argo NoC [7]. They implemented send and receive operations as well as barriers and broadcasting. In our paper, we focus on collective operations and their impact on timing analysis. Moreover, we develop a concept for the complete timing analysis of MPI programs. Furthermore, we apply it on a complete benchmark and compare two different variations of time division multiplexing (TDM).

2.2 PaterNoster NoC and our Implementation

For being able to estimate the WCET of an MPI operation, it is required that the underlying NoC ensures upper bounds for communication. In our paper, we use the PaterNoster NoC [10] which fullfills this requirement by providing *guaranteed service* (GS) with TDM [11], see details in Section 2.3.

The nodes in our NoC are arranged in a quadratic torus and connected via unidirectional X rings (horizontal) and Y rings (vertical). Each node consists of a processing element (core), a sufficiently sized send/receive buffer and a lightweight router. Data exchange takes place via *flits* that are sent from one node to another over the NoC. Because the information where a flit is to be sent is included seperately, there is no need of a head flit. Flits are 32 bits wide and are forwarded instantly without buffering. They first take the X direction and then the Y direction (xy-routing). When flits change their direction from X to Y, they are stored in a so-called corner buffer until it is the right time to leave the node. The right time to leave and arrive at buffers is determined by a schedule, see the following Section 2.3.

The goal of our implementation is to abstract communication in a parallel program: instead of making a detailed WCTT analysis for every program again, the already known WCET estimates of MPI collectives may be used. The WCET estimate of a program can then be determined by adding the WCET estimates of MPI operations to the WCET estimates of sequential parts. Because all nodes execute the same sequential code and we take its WCET estimate, it can be assumed that they are all finished when reaching a MPI operation. However, there are some restrictions to enable our implementation to stay general, e. g. no derived data types are allowed and we assume that no flit gets lost. Sometimes, more flits have to be sent at collective operations than with direct communication. One example are acknowledgement flits, which contain no real data, but only the information that a communication partner is ready.

2.3 Scheduling

TDM means that shared resources are available for each requester for a fixed time interval. Each of them has its own time slot – these are ordered in a way that no conflicts can occur. At a NoC this means that for each pair of senders and receivers it is clear when their flit is at which location in the NoC – ensuring that no other node will place a flit there at the same time. This enables estimating a WCTT – first, each participant has to wait until its time slot is available, then the flit can be transmitted. The time intervals and their order form one *round* of a TDM schedule.

There are two types of schedules: custom and generic schedules. Custom schedules are computed for the specific configuration of applications on a core. They feature a good (worst-case) performance, but each time something is changed, they have to be recomputed since a change may result in a conflict. Generic schedules are application-independent – they describe a regular pattern: when is each node allowed to send flits? How many? Which nodes are allowed for receiving these flits?

The (worst-case) performance of generic schedules is worse than that of custom schedules. However, it is possible to give general statements and changes are easier to handle than with custom schedules. In our case, we make general statements about the timing behaviour of MPI collective operations. Our timing estimation applies for any application utilising this collective operation, presuming that the same NoC and schedule is used.

Schoeberl et al. propose a generic All-To-All schedule (AA) [13]: within one period of the schedule, each node is allowed to send flits to any other node. However, each node is allowed to send at most one flit to the same node. In the worst-case this means that all nodes send one flit to each other node. Mische et al. propose a One-To-One schedule (11) amongst others [11]: each node is allowed to send and receive at most one flit in one period. This is much stricter than All-To-All, but results in shorter periods, when not all nodes participate. Furthermore, the worst-case is the same as the average case: each node sends and receives one flit. Sending several flits takes several rounds. A detailed comparison of different generic schedules for MPI collective operations takes place in [15].

3 Timing Analysis of MPI Collectives

3.1 Setting for the Analysis

Generally, our approach is platform-independent. However, to get concrete numbers, we make a timing analysis on a custom platform. It is composed of 16 nodes connected via the PaterNoster NoC and arranged as a 4x4 unidirectional torus. The NoC has been already described in Section 2.2. Each node consists of a simple core with ARM instruction set, 5 stage pipeline, no caches and 10 cycles memory access latency to the local memory. We apply a 1:1 mapping meaning that each core executes one thread.

We utilise the static WCET analysis tool OTAWA [3] for the sequential program parts. 3 cycles are assumed for the assembler instructions for sending and receiving a flit. Another 4 cycles are assumed for the time between execution of the assembler instructions and availability of the flits at the NoC buffer (and vice versa)[2]. This time can be changed at any time by adjusting the parameter t_{Buf} at the equations in the rest of this paper. For communication between nodes we considered two schedules: All-To-All as described by Schoeberl et al. [13] or One-To-One as described by Mische et al. [11]. Their WCTTs are described by the following two equations, where n is the dimension of the NoC (4 in our case), f is the number of flits to be transmitted and χ is the number of participating nodes (excluding the master node). Details can be found in [11, 15]:

$$WCTT_{All-To-All} = \frac{n^2 \cdot (n+1)}{2} \cdot f + \frac{n^2}{2} + 2n \tag{1}$$

$$WCTT_{One-To-One} = n \cdot \chi \cdot f + 2n \tag{2}$$

[2] This time is very short because our group develops hardware support for fast message passing to substitute shared memory synchronisation.

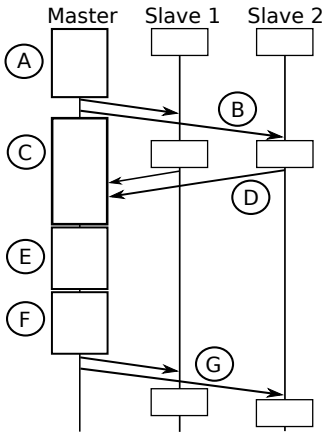

Figure 1 MPI_Allreduce with one master and two slave nodes.

Applying these equations with parameters later needed in this paper gives following numbers: When 1 flit is to be transmitted in a 4x4 NoC with All-To-All schedule, we get a WCTT of 56 cycles. With the same schedule, the WCTT is 136 cycles for 3 flits, 616 cycles for 15 flits or 14 056 cycles for 351 flits. Utilising the One-To-One schedule, 1 flit can be transmitted to or from 2 participating nodes with a WCTT of 16 cycles or 351 flits in 2 816 cycles. When χ and f are both 15 at the One-To-One schedule, the WCTT is 908 cycles, while it would be 44 cycles when they are both 3. These numbers will be used later in our analysis.

3.2 MPI_Allreduce

MPI_Allreduce and its variations are defined as "global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group" [4]. Thereby, all participating nodes send their values to the master node, which combines them with the given operation. In the benchmark example in Section 4, this operation is summing up the values. When the master node has finished collecting and totalising, it sends the result to all participating nodes.

3.2.1 MPI_Allreduce: Structure

The implementation of MPI_Allreduce is structured as follows (letters correspond to Figure 1):

(A) First, there is a short initialisation phase.
(B) Then, the master node sends an acknowledgement flit to all participating nodes.
(C) While the flits are on their way, the master node initialises some data structures preparing receiving of the values and the operation to be performed on them. At the slave nodes, there is a small sequential code after receiving the acknowledgement flit.
(D) Now, the slave nodes send their values to the master node. When more than one value should be sent, slave nodes go on sending without waiting for further acknowledgement flits from the master node.
(E) The master node gathers the values sent from the slave nodes and collects them in an array. Afterwards, the master node copies its own values to the array.
(F) Then, the collective operation is applied on the collected values (e. g. summing up values).
(G) Finally, the result of the operation is broadcasted to all participating nodes.

Table 1 Execution steps and their estimated WCET contribution to MPI_Allreduce.

Step	Estimated WCET contribution
A	73
B	12χ (local processing of broadcast)
C,D	$\max(23 + 6n^2 + 11\chi, 2 \cdot (t_{transm,\chi} + t_{Buffer}) + 24)$
D	$(f-1) \cdot \max(35\chi, t_{transm,\chi})$ (when there is more than one value)
E	$35\chi + 15 + 32f$ (processing and copying own values)
F	$42 + (\chi + 1) \cdot (94 + 23f)$ (arithmetic op.) or $42 + (\chi + 1) \cdot 41$ (bitwise op.)
G	$14 + f \cdot (11 + 12\chi) + f \cdot t_{transm,\chi} + t_{Buffer} + 35$

3.2.2 MPI_Allreduce: Timing Analysis

There is one prerequisite for the timing analysis: data structures have to be allocated statically. Since MPI_Allreduce gets the data structures handed over as calling parameter, this is left to the program.

Table 1 illustrates the execution steps of MPI_Allreduce and their contribution to the WCET estimate: After the initialisation, the broadcast of an acknowledgement flit is prepared at the master node (A). This is sent out and processed by the slave nodes (B), who reply with the (first) value to be sent (D). Meanwhile, the master node prepares data structures needed for the receiving and the collective operation (C). When C is finished before D, it has to wait for D and vice versa. Therefore, the maximum of C and D has to be taken into account for the WCET estimate.

In the case that more than one value is to be sent, the maximum of the time to receive flits (processing received values at the master node and store them in an array) and the time to transmit flits has to be determined (step D). At step E, the last received values are stored in the array and the values from the master node are appended. Afterwards, an arithmetic operation (e.g. SUM, MIN, MAX) or a bitwise operation (e.g. AND, OR, XOR) is applied on the collected values (F). Finally, broadcasting of the results is prepared and performed, followed by postprocessing in step G.

The variables in Table 1 were already described in Section 3.1. Additionally, two types of transportation times exist: $t_{transm.,\chi}$ is the time to get χ flits transported from all nodes to one node or vice versa, while t_{Buf} is the time flits need to pass from the NoC to the pipeline and vice versa. We assume 4 cycles to move from the sender's core to the NoC router and 4 cycles to move from the receiver's NoC router into the pipeline, which are together 8 cycles.

Alltogether, the WCET estimate of MPI_Allreduce can be rewritten as the following equation (the elements from Table 1 are summarised and transformed):

$$\begin{aligned} WCET_{AR}(f,\chi) = {} & 273 + 35f\chi + \max(23 + 6n^2 + 11\chi, 24 + 2(t_{transm,\chi} + t_{Buf})) \\ & + 141\chi + (f-1)\max(35\chi, t_{transm,\chi}) + (66 + t_{transm,\chi})f + t_{Buf} \end{aligned} \tag{3}$$

Utilising the numbers from Section 3.1, WCET estimates for several scenarios can be computed. For example, the WCET estimate of MPI_Allreduce is 6 698 cycles for 2 flits to be transmitted to or from 15 nodes when the All-To-All schedule is used, while it is 8 158 cycles with the One-To-One schedule. When f is 351 and χ is 3, All-To-All's WCET estimate is 156 373 cycles while One-To-One's WCET estimate is 113 073 cycles. Due to traversal times which are influenced by group sizes, One-To-One scheduling is better than All-To-All scheduling in the second case.

Table 2 Execution steps and their estimated WCET contribution to MPI_Sendrecv.

Step	Estimated WCET contribution
Initialisation	20
acknowledgement sender-receiver	$\max(5, t_{transm,1} + t_{Buf})$
time between acknowledgements	7
acknowledgement receiver-sender	$\max(5, t_{transm,1} + t_{Buf})$ (only if sender != receiver)
initialisation for sendrecv-loop	15
sendrecv-loop	$15 + \max(f \cdot 32, t_{transm,f}) + t_{Buf}$
postprocessing, finish function	51

3.3 Analysis of MPI_Sendrecv

Analogous to MPI_Allreduce, we analyse the operation MPI_Sendrecv. Its purpose is to send and receive messages at the same time. The communication partners for sending and receiving do not need to be the same. Table 2 illustrates the parts of MPI_Sendrecv and shows their calculated WCET estimates.

f is the maximum of the number of flits to be sent and received. $t_{transm,f}$ is the time needed to transmit f flits through the NoC, while $t_{transm,1}$ is the time needed to transmit 1 flit through the NoC.

The process of MPI_Sendrecv works as follows: After the initialisation, the sender sends an acknowledgement flit to the receiver, who waits for it. When the communication partners for sending and receiving are different, the other core also acknowledges the communication. After finishing acknowledgements, the loop for sending and receiving data is first prepared, then executed. Finally, some postprocessing takes place before the function is finished.

Since transmission and buffer times are expected to be greater than five, the equation may be written as shown in Equation (4):

$$WCET_{SR}(f) = 108 + 2 \cdot (t_{transm,1} + t_{Buf}) + \max(f \cdot 32, t_{transm,f}) + t_{Buf} \tag{4}$$

Again, the numbers from Section 3.1 can be used in this equation. For sending and receiving 351 values to/from two different nodes, the WCET estimate of MPI_Sendrecv is 14 300 with the All-To-All schedule and 11 396 with the One-To-One schedule.

3.4 Differences at varying configurations

While sequential parts always remain equal, the impact of communication parts changes with the size of the NoC: With increasing NoC size, communication times increase, too. Furthermore, the impact of the schedule also increases.

On the other side, a timing anomaly can occur at a small NoC: when the transmission is faster than new flits are provided by the core, it might have to be assumed that a flit misses a round of the schedule and has to wait until the next round is carried out (each round one flit can be transported). This leads to a "step" at the admission time of the flit, which could cause disadvantageous configurations of small NoCs being slower than a little bit larger NoCs.

Attention has to be paid at the receive buffer: a core is stalled when the send buffer is full, which means that everything still works fine because the WCET is driven by the WCTT. However, all schedules rely on free capacity at the receive buffer. When it is full, receiving of flits does not follow the schedule anymore and blows up the WCET. It could also mean that

estimating a WCET is not possible anymore. At our analysis, we assume that buffers are large enough.

4 Case study: Timing Analysis of the CG benchmark

Utilising the results from the previous Section, we analyse the conjugate gradient (CG) benchmark, which is taken from the NAS Parallel Benchmark Suite[3] [1, 2]. It is described as following: "a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, using unstructured matrix-vector multiplication" [1]. The NAS Parallel Benchmarks were developed for highly parallel systems. Therefore, they seem to be good benchmarks for embedded real-time multicores with distributed memory.

For the analysis, we decided to analyse a class S CG benchmark, which is the smallest class: The matrix size is $1400 \cdot 1400$, there are 7 nonzero values per row and 15 main benchmark iterations take place[4]. In the benchmark, the matrix is divided into equal sized blocks and each block is assigned to one core for computation.

Several changes had to be done for analysis: first, the benchmark had to be ported from FORTRAN 77 to C99. Then, we had to ensure that no data structures are allocated dynamically. Since OTAWA's support for floating point is not yet available for the ARM instruction set, we used integers instead of doubles[5].

The structure and estimated WCETs' contributions of the benchmark are illustrated in Table 3. Thereby, we did not analyse the initialisation of the benchmark (which is not included in the table), but the parts which are intended for benchmarking a system. This means the steps 1 to 18 have to be executed: After a sequential part at the beginning and one execution of MPI_Allreduce, a `for` loop is executed, which is iterated 15 times (estimated WCETs in the third column are for one iteration, those in the last column for 15 iterations). Afterwards, four sequential parts and communication parts alternate, before the end of the main iteration loop is reached. In the right column of Table 3 it is shown how much each step totally contributes to Equation 5.

Altogether, CG benchmark's estimated WCET can be summarised as follows:

$$\begin{aligned} WCET_{cg} = 1\,896\,959 + WCET_{AR}(2,15) + 17 \cdot WCET_{AR}(1,3) \\ + 16 \cdot (WCET_{AR}(351,3) + WCET_{SR}(351)) \end{aligned} \tag{5}$$

1 896 959 is the sum of the sequential parts from Table 3, where the for loop was respected with 15 iterations. Most of the variables needed to get a total WCET estimate were already computed throughout the previous Section 3. The only missing number is $WCET_{AR}(1,3)$, which is 1 323 with the All-To-All schedule and 1 071 with the One-To-One schedule. The numbers can be placed in Equation (5) and lead to following results:

$$WCET_{cg,AA} = 1\,896\,959 + 6\,698 + 17 \cdot 1\,323 + 16 \cdot (156\,373 + 14\,300) = 4\,656\,916 \tag{6}$$

$$WCET_{cg,11} = 1\,896\,959 + 8\,158 + 17 \cdot 1\,071 + 16 \cdot (113\,073 + 11\,396) = 3\,914\,828 \tag{7}$$

[3] `http://www.nas.nasa.gov/Software/NPB/`.
[4] We only analyse 1 iteration – the result may be multiplied by 15 to get the final result.
[5] This leads to loss of precision, but all computational steps keep the same. We do not focus on analysing all sequential instructions precisely, but to demonstrate the possibility of analysing parallel program structures.

Table 3 Parts of the main iteration loop and their estimated WCET contribution.

Step	Description	Est. WCET contrib.	Contrib. to Equation 5
1	Start of main iteration loop	15 929	15 929
2	MPI_Allreduce	$WCET_{AR}(1,3)$	$WCET_{AR}(1,3)$
3	**Begin of for loop (15 iterations)**	100 490	$15 \cdot 100\,490 = 1\,507\,350$
4	MPI_Allreduce	$WCET_{AR}(351,3)$	$15 \cdot WCET_{AR}(351,3)$
5	Sequential part	2 480	$15 \cdot 2\,480 = 37\,200$
6	MPI_Sendrecv	$WCET_{SR}(351)$	$15 \cdot WCET_{SR}(351)$
7	Sequential part	10 749	$15 \cdot 10\,749 = 161\,235$
8	MPI_Allreduce	$WCET_{AR}(1,3)$	$15 \cdot WCET_{AR}(1,3)$
9	**End of for loop (15 iterations)**	3 792	$15 \cdot 3\,792 = 56\,880$
10	Sequential part	100 513	100 513
11	MPI_Allreduce	$WCET_{AR}(351,3)$	$WCET_{AR}(351,3)$
12	Sequential part	2 480	2 480
13	MPI_Sendrecv	$WCET_{SR}(351)$	$WCET_{SR}(351)$
14	Sequential part	3 180	3 180
15	MPI_Allreduce	$WCET_{AR}(1,3)$	$WCET_{AR}(1,3)$
16	Sequential part	8 142	8 142
17	MPI_Allreduce	$WCET_{AR}(2,15)$	$WCET_{AR}(2,15)$
18	End of main iteration loop	4 050	4 050
odd	Sum of sequential parts		1 896 959

With the All-To-All schedule, one main iteration loop of the CG benchmark has an overall estimated WCET of 4 656 916 cycles, while it is 3 914 828 with the One-To-One schedule. The result of the One-To-One schedule is better than with the All-To-All schedule, because its periods are shorter when only a part of the nodes participates at communication with one node. However, for large groups the All-To-All schedule outperforms the One-To-One schedule.

Furthermore, it can be seen that the communication times are very large compared to the sequential program parts. This is caused by the communication intensive program structure, mainly influenced by the operations where 351 values are exchanged. These operations are executed 16 times.

5 Conclusion and Outlook

We presented our idea that collective (MPI) operations are beneficial for timing analysis of parallel distributed memory platforms with message passing communication. Timing analysis is performed of the two MPI collective operations MPI_Allreduce and MPI_Sendrecv for a platform with simple ARM cores connected via the PaterNoster NoC. Afterwards, we used these results to make a timing analysis of the CG benchmark, which does a lot of inter-core communication to move data and coordinate computation.

Maybe there are better implementations exhibiting lower WCET bounds for the MPI collectives. With this paper, we made the first step to show the feasability that MPI collectives could be one way to enable timing analysis for parallel platforms. Our next step will be to tighten the WCET estimate with (i) an improved MPI implementation, (ii) better workload distribution within MPI collectives and (iii) optimised hardware support. Furthermore, we see the need to implement and analyse more collectives, as well as making the results open source at `https://github.com/unia-sik`.

References

1 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. doi:10.1177/109434209100500306.

2 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing'91, pages 158–165, New York, NY, USA, 1991. ACM. doi:10.1145/125826.125925.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *LNCS*, pages 35–46. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-16256-5_6.

4 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015.

5 M. Frieb, R. Jahr, H. Ozaktas, A. Hugl, H. Regler, and T. Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *International Journal for Parallel Programming*, 2016. doi:10.1007/s10766-016-0432-7.

6 A. Kanevsky, A. Skjellum, and A. Rounbehler. MPI/RT – an emerging standard for high-performance real-time systems. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 157–166, 1998. doi:10.1109/HICSS.1998.656130.

7 E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. doi:10.1109/TVLSI.2015.2405614.

8 B. Lisper. Towards Parallel Programming Models for Predictability. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23, pages 48–58, Dagstuhl, Germany, 2012. doi:10.4230/OASIcs.WCET.2012.48.

9 S. Metzlaff, J. Mische, and T. Ungerer. A real-time capable many-core model. In *32nd IEEE Real-Time Systems Symposium: WiP Session*, pages 21–24, Vienna, Austria, 2011. URL: http://www.cs.wayne.edu/~fishern/Meetings/wip-rtss2011/WiP-RTSS-2011-Proceedings-Post.pdf.

10 J. Mische and T. Ungerer. Low power flitwise routing in an unidirectional torus with minimal buffering. In *Fifth International Workshop on Network on Chip Architectures*, NoCArc'12, pages 63–68, New York, NY, USA, 2012. ACM. doi:10.1145/2401716.2401730.

11 J. Mische and T. Ungerer. Guaranteed service independent of the task placement in nocs with torus topology. In *22nd International Conference on Real-Time Networks and Systems*, RTNS'14, pages 151:151–151:160, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659804.

12 C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 90–100, Dagstuhl, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.90.

13 M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Sixth IEEE/ACM In-*

ternational Symposium on Networks on Chip (NoCS), pages 152–160, May 2012. doi:10.1109/NOCS.2012.25.

14 A. Skjellum, A. Kanevsky, Y. Dandass, J. Watts, S. Paavola, D. Cottel, G. Henley, L. S. Hebert, Z. Cui, and A. Rounbehler. The Real-Time Message Passing Interface Standard (MPI/RT-1.1). *Concurrency and Computation: Practice and Experience*, 16(S1):i–322, 2004. doi:10.1002/cpe.744.

15 A. Stegmeier, M. Frieb, J. Mische, and T. Ungerer. WCTT bounds for MPI Collectives in the Paternoster NoC. In *14th International Workshop on Real-Time Networks (RTN)*, Toulouse, France, 2016.

16 R. B. Sørensen, W. Puffitsch, M. Schoeberl, and J. Sparsø. Message passing on a time-predictable multicore processor. In *IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 51–59, April 2015. doi:10.1109/ISORC.2015.15.