Journal of Information Processing Vol.20 No.3 559–569 (July 2012)

[DOI: 10.2197/ipsjjip.20.559]

Regular Paper

Implementation and Verification of Concurrent Sorting Algorithms with CSP based Architecture

Yuki Hasegawa[1,a)] Yoshinao Isobe[2,b)] Kazuhito Ohmaki[1,c)]
Hideki Mori[1,d)] Kensei Tsuchida[1,e)] Yasunori Shiono[1,f)]

Received: September 15, 2011, Accepted: February 3, 2012

Abstract: Communicating Sequential Processes (CSP) based architecture is regarded as a useful method in the development of concurrent embedded systems. Products around us are embedded in many computer systems. Concurrent processing by software is necessary in multi-core and multi-processor environments to make more effective use of hardware resources. There is strong demand for hierarchy, resource constraints, and safety for implementation of embedded systems. We implemented a sorting model as a concurrent system in an experiment. We tried to design, implement, and verify concurrent sorting model with CSP based architecture. In this study, we try to parallelize of sorting as the subject of embedded systems for implementing. Because sorting has been widely studied, it is suitable as the subject of parallelization. We also evaluated the system. We will consider the usefulness of CSP, which we present in this paper, using examples of development.

Keywords: concurrent sorting, CSP architecture, embedded systems, formal methods

1. Introduction

The usefulness of the Communicating Sequential Processes (CSP) [1], [2] architecture has been focused on in the development of concurrent embedded systems.

Embedded systems are computer systems used to control components and achieve specific functions, and they have been built into consumer electronics and industrial equipments. Embedded technology is also required in various fields. Concurrent processing by software is necessary in multi-core and multiprocessor environments to effectively use hardware resources. There has been strong demand for hierarchy, resource constraints, and safety in the requirements [4] to implement embedded systems.

We implemented an embedded system with methodology based on CSP in experiments in this study as an example of developments. We tried to parallelize sorting model in this study to implement it in embedded systems. Sorting [9] is done by algorithms that relocate sets of data according to certain rules. Sorting has clearly defined requirements for internal processes and end states. Sorting is therefore suitable for evaluating and verifying the parallelization of algorithm.

We tried to develop parallel systems based on CSP as an example of sorting model. Asynchronous processing by concurrent systems is easy to develop in the hierarchy of module units. Concurrent systems are easy to construct with CSP because development processes that implement modules cooperate while they are synchronously communicating. Modules can be expected to reduce development efforts because they are highly reusable and able to be developed in parallel. An abstract model of the system by using CSP (CSP model) is easy to verify with a model checker. Verification certifies that the CSP model (concurrent specifications) and original specifications (sequential) are equivalent, and it ensures the safety of implemented model. The hardware that implements the CSP model can be selected with XMOS [8] and Verilog [5].

We tried to implement two sorting models in the experiments. The first model (Star model) was a hierarchical model of the system by using functions. The second, Ring model, was a model that took resource constraints into consideration. Concurrent processing generally makes it difficult to ensure safety, due to deadlocks and livelocks.

Verification of the model with CSP ensures the safety of the implemented model. We thought that methodology based on CSP would be useful to develop of concurrent embedded systems after the experiments. We considered the usefulness of CSP by develop a CSP based case, which is discussed in this paper.

2. Background

2.1 Embedded Systems

An embedded system is one that controls the components and specific functions and is built into industrial equipments and consumer electronic devices [3], [4], [5], [6].

Product life cycles are currently being shorted, and the periods from development to verification are being completed within

1 Toyo University, Kawagoe, Saitama 350–8585, Japan
2 National Institute of Advanced Industrial Science and Technology, Ikoma, Nara 630–0192, Japan
a) gz090012@gmail.com
b) y-isobe@aist.go.jp
c) ohmaki@toyo.jp
d) mori@toyo.jp
e) kensei@toyo.jp
f) shiono@toyo.jp

© 2012 Information Processing Society of Japan

about three months.

Four requirements are needed to implement embedded systems [3], [4], [5], [6].

- Concurrency
  Multi-core and/or multiprocessors are becoming dominant in the architecture of processors as a solution to the limits in circuit line width, increased generation of heat, and clock speed limits.
  Therefore, it is necessary to implement applications by using methods with parallelism descriptions.
- Hierarchy
  System modules are arranged in a hierarchal fashion in main systems, subsystems, and sub-subsystems. Diversity and recycling can be improved, and reducing the number of development processes as much as possible.
- Resource Constraints
  It is necessary to comply with the constraints of built-in objects like memory and power consumption.
- Safety and Reliability System failure is a serious problem causing severe damages and fatal accidents. It is extremely important to guarantee the safety of the system.

2.2 Formal Methods

Concerns regarding the reliability and safety of software and hardware modules have recently intensified, and formal methods have therefore attracted a great deal of attention.

Formal methods [7] are useful for system development, especially software, and they have a background of mathematical logic. It is possible to systematically evaluate the accuracy of designs by describing design objects using methods that are based on approaches that are mathematically clear and rigorous. Therefore, the developed system can be guaranteed to be very safe and reliable.

3. CSP

Communicating Sequential Processes (CSP) is one of the most famous formal methods and it involves typical process algebra based on synchronous communications [1], [2]. Process algebra is a theory used to formally describe and analyze concurrent processing.

The five main features of CSP are:

- Sequential processing description
  Serial processing is described as an ordered set of processes.
- Concurrent processing description
  Concurrent processing is described as a synthesis of processes in parallel.
- Event prefix
  An event means actions/interactions for processes such as interprocess communication.
- Message passing
  The process exchanges messages between channels on the event.
- Selective description Behavior selected with the event is described.

The ratio of the execution workload/time (e) and traffic load/time (c) is important; this corresponds to the ratio of the execution throughput (E) and communication throughput (C) of a node. It indicates the performance of concurrency.

We use expressions to specify processes in CSP. These processes can verify whether several behavioral equivalences among processes are satisfied or not, by re-writing these expressions based on the algebraic rules of the CSP. For example, we can write a sequential process, SEQ, in CSP as:

SEQ = call?x → ret!(f(x)+g(x)) → SKIP

Where **c!v** represents an output event and **c?x** represents an input event. Here, → is the execution of an event and **SKIP** is a successful termination process. That is, process **SEQ** receives a value from **x** through a channel call. After event **call?x**, value **f(x)+g(x)** is sent through channel **ret**. After event **ret!(f(x)+g(x))**, **SEQ** is successfully terminated by **SKIP**.

We expect high-speed execution of concurrent processes when they are running on a real multi-core parallel environment. However, the behavior of these processes is more complicated than that with a sequential environment. We can verify several equivalent relations between processes using CSP. In other words, we can observe the complicated behavior of concurrent processes by replacing them with equivalent sequential processes. These equivalent relations are verified many times before actual implementation. We can then finally install equivalent concurrent processes on an actual system.

4. CSP Platform

4.1 Architecture

The execution platform of the CSP model is a processor set, where all processors are connected to one another with synchronization channels. Channel connection involves five models.

- One to one connection model
- One to multi connection model
- Multi to multi connection model
- Mesh connection model
- Network model

The Mesh connection model is outlined in **Fig. 1**.

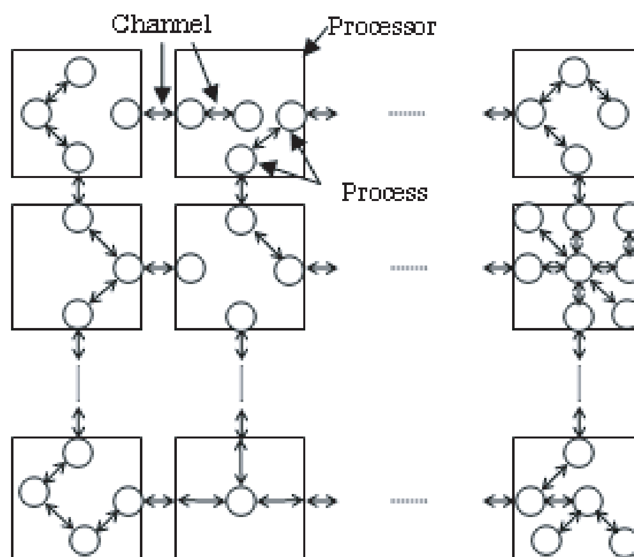


Fig. 1 CSP platform (grid).

© 2012 Information Processing Society of Japan

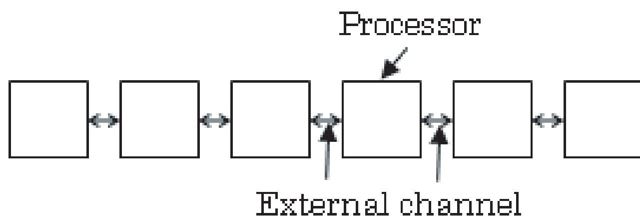


Fig. 2 CSP platform (line).

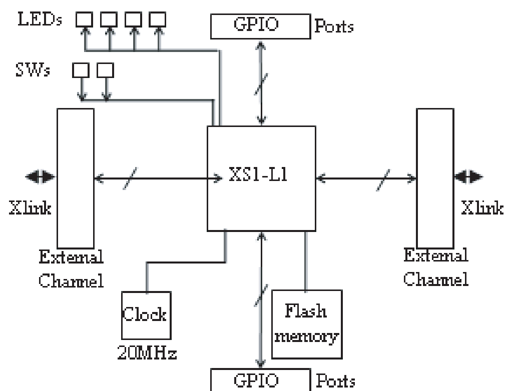


Fig. 3 CSP platform (block diagram: XK-1).

The platform that we used for the experiments is the mesh model with the series connection of six processors shown in **Fig. 2**. We used a XMOS XS1-L1 [8] as the processor element. The XS1-L1 included synchronous channel communication and event driven mechanisms to effectively achieve CSP. We implemented our CSP platform using 6 XK-1 boards that included the XS1-L1 and peripherals.

The XK-1 development board included LEDs, push-buttons, and an SPI flash memory, which contained the control programs. Two or more XK-1 development boards could be connected to one another, using two 20-pin XSYS connectors. Then they were integrated and controlled by one program. And the logical connection was represented by an XN file. It was also possible to connect external components such as LEDs, motors, and sensors with two 16-pin IDC connectors. Therefore, an environment for XMOS development such as mobile devices and robots could be prepared.

Three hardware resources must be considered in the XMOS processor, which are necessary for designing the system.

- Eight processes/one core
- 32-channel ends/one core
- One XS1-L1 processor in an XK-1 development board

A maximum of eight concurrent processes is possible with one processor unit. Therefore, it is possible to concurrently connect more XK-1 development boards by running the processors in two or more cores, using this method of connection.

Figure 3 shows a block of an XK-1 development board. We used the XC language developed by XMOS for programming.

4.2 CSP Oriented Language

The CSP oriented languages were Occam, JCSP, and XC [8] developing language.

We used the XC developing language that XMOS Ltd. developed for the XMOS processor for evaluation.

The XC developing language is an extended version of C, in which I/O functions (sending and receiving using channels or ports), time management, and concurrent processing (par sentence) functions are added in addition to standard C control statements such as “while,” and “do-while.”

XMOS development tools, which integrate C and XC compilers, a simulator, and a debugger, enable systems and algorithms to be developed using parallelism, concurrent and real-time programming, CSP-based communications, and event-driven control.

5. Concurrent Method of Sorting model

5.1 Sorting and Implementation of Concurrent Programming

Sorting involves an algorithm that relocates a set of data according to certain rules. Computational complexity and sorting differ according to the algorithm used to implement them. Sorting has also clearly defined requirements for internal processes and states. These are very easy to assess. In addition, well-known algorithms such as insertion sort have already been analyzed in many studies.

We tried to implement concurrency, verification, and evaluation with sorting as the subject in this experiment. Sorting with a clear definition of the algorithm made it easy to evaluate performance in implementing concurrency. We tried to develop a parallel sorting model by focusing on resource constraints and hierarchy in this experiment.

5.2 Parallel Split Sorting Model

We developed a parallel sorting model, which we called the “Parallel Split Sorting Model.” This model is a method of splitting data that are to be sorted. It is intended to reduce the computational time it takes for sorting without splitting data. This model’s main features are “split data” and “parallel sorting.” The sorting algorithm can be used to implement existing algorithms such as insertion sort. The computation time data are reduced by splitting data. These data are then sorted in parallel.

5.3 Parallel Split Sorting Algorithm

The algorithm for this model involves six steps:

1. Input the test data of N.
2. The data set is split on M nodes, and transferred to nodes.
3. Each node is sorted in concurrent processing.
4. Each node exchanges data between neighboring nodes if necessary.
5. Repeat the exchange until data do not need to be replaced.
6. Finally, we combine data from each node, and use this algorithm to sort all data.

This sorting uses an existing sorting algorithm such as insertion sort.

The algorithm in **Fig. 4** is composed of two types of processes: IO and Sort. Each process is assumed to operate asynchronously.

IO proc is a process that inputs unsorted data, monitor the exchange flow, and outputs sorted data. This process is the only one in the program. After “input Test data” has finished, the **transfer data** subsets of data transfer them to **Sort proc** processes while

© 2012 Information Processing Society of Japan

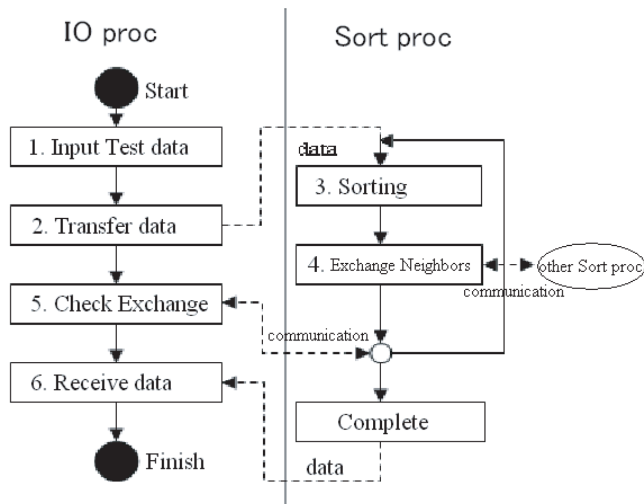


Fig. 4 Split sorting flow.

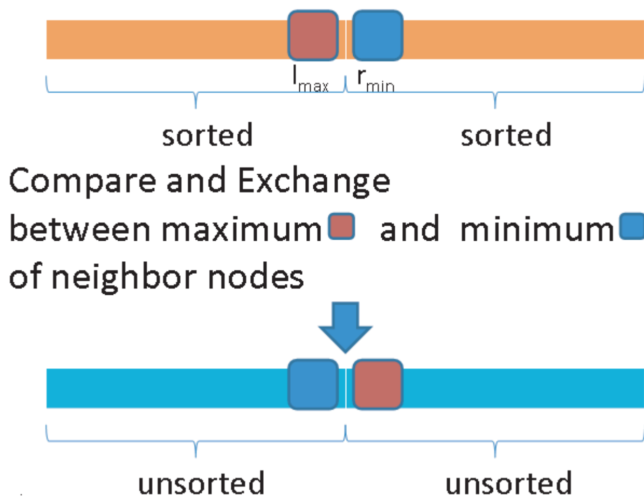


Fig. 5 Exchange neighbor nodes.

splitting the data.

Sort proc is a process that sorts the subset of data and neighbor exchange. Several sort processes are implemented and connected in one model. The system updates the data with each process running concurrently.

The **Sort proc** process finishes by exchanging a neighbor node's data in communication to inform whether exchange has occurred or not. If a node's data can be exchanged, the system sorts to start next sort.

If a node's data cannot be exchanged at all nodes, the system completes sorting and transfers data to the **IO** process. There is always one **IO** process in the implementation. However, the Sort process splits the number of nodes M.

The main advantage of this model, i.e., parallel processing by splitting the sorting part, is that computation time can be reduced.

We assumed that data would be sorted in ascending order from left to right, as shown in **Fig. 5**. Each node M (Sort process) is located in a row from left to right.

First, the data that are a subset of the original data are split, and each node is assigned by each transfer. Each node sorts data and each node's data are sorted. These data are combined but the data sets are unsorted.

Table 1 Property of bitonic sort.

	Value	Remarks
Number of processor	M	M must be 2^x (x>0).
Time complexity	$\Omega(log^2N)$	It is minimum when N/M equals 1.
Space complexity	$O(N)$	

Table 2 Property of parallel split sorting.

	Value	Remarks
Number of processor	M+1	M>1,and '+1' is IO process.
Time complexity	$O(N^2)$	Case of using insertion sort.
Space complexity	$O(N)$	

Therefore, we focused on the data of boundary (l_{max}, r_{min}) between two sorted neighbor nodes. Then, we compare l_{max} (maximum data on the left node) and r_{min} (minimum data on the right node). As a result, if $l_{max} > r_{min}$, we exchange these data because this is not in ascending order in this part of data. If $l_{max} < r_{min}$, the data will be completely sorted only in them because this is in ascending order and the data in each node are sorted.

In other words, we compare the data between neighbor nodes, exchange them when the order relation is correct, or do not exchange them when the order relation fails. Finally, if there are no exchanges between any neighbors, the data will complete sorting by combining all data nodes.

5.4 Comparison with the Other Works

5.4.1 Bitonic Sort

Bitonic sort [12], [13] is an efficient sorting network.

Sorting network is configured in Wire and Comparator. Wire is placed in more than two lines and it is used to carry data. Comparator is connected between the two wires and it compares and exchanges data at the connecting point. When Sorting network is implemented as a parallel sorting, its wire is defined in process (processor). If the number of data handled by the processor is only one, the behavior of the comparator is simple to interpret. However, one element of data in a process is unrealistic in a limited environment such as an embedded system. In implementation of Bitonic sort, the number of processes is needed $2^x(x > 0)$.

A narrow Bitonic sort can sort Bitonic sequence in $O(log(p))$ steps. In order to convert the general sequence to Bitonic sequence will require the necessary steps to sort half of the input data.

5.4.2 Advantage of Parallel Split Sorting

Our Parallel Split Sorting model provides a connection between the process, which involves sorting in parallel, and a method to determine the sorting is complete. The process of sorting algorithms can use existing, verified sorting algorithm.

Our model is a realistic implementation model, and can handle a static change to the number of divisions (the number of parallel sorting processes). In implementation by our model, the number of processes is not limited to $2^x(x > 0)$.

In **Table 1** and **Table 2**, we show the comparison of number of processors, time complexity, and space complexity in the proposed method and Bitonic sort. N and M are constants. N is the number of target data. M is the number of division (the number of parallel sorting process).

© 2012 Information Processing Society of Japan

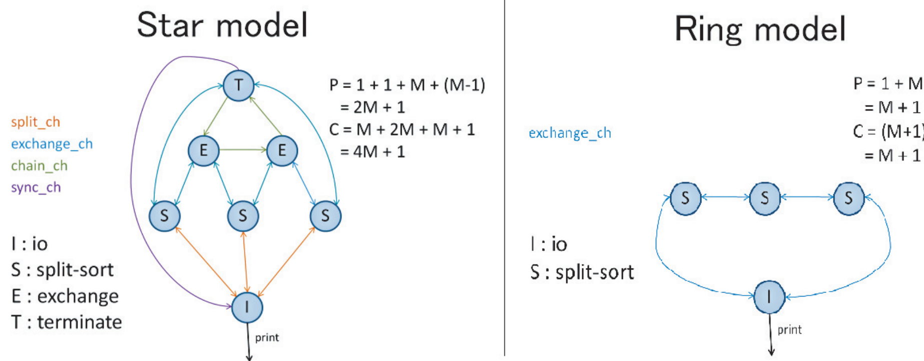


Fig. 6 Concurrent sorting models.

6. Experiment

6.1 Parallel-Split-Sorting Model

6.1.1 Design of Parallel-Split-Sorting Model

We developed two models, which we called the star model and the ring model in the experiment.

The star model focused on hierarchy and it implemented processes that were split by functions.

The ring model focused on resource constraints and reduced the number of processes by improving the star model.

Both models are shown in **Fig. 6**, where the circles represent processes, the lines represent channels, and vectors represent the direction communication is sent. The P and C correspond to the number of processes and channels, when the number of original data is split. The implemented sort algorithm is an insertion-sort type.

6.1.2 Implementation of Method

We implemented these models on the XK-1 development board using XC language and an XC compiler. The maximum number of enabled cores was six (Core[0-5]) in the experiment with a maximum of 48 processes since six XK-1 were used. However, the number of channels and memories were constrained.

The results we obtained from the experiment will be discussed in the next section.

6.2 Experiment on Our Star Model

6.2.1 Design

We implemented **Star-par-split-sort** for Parallel Split Sorting with star model (Fig. 6 (Left)).

The star model is composed of IO process (**I**), sorting (**S**), data exchange between neighbor nodes (**E**), and checking exchanges (**T**). This design is simple.

This model has a star formation because **I** is mainly connected to the star. Therefore, the number of **split_ch** channels connected to **I** depends on the number of splits.

First, **I** inputs the test data, and transfers the split data to S with **split_ch**.

S sorts the data when it receives them from **I**. To exchange neighbor nodes, **S** transfers the maximum and minimum data to **E** (exchange process) or **T** (terminate process) connected with **exchange_ch**.

E receives data from the two **S**s connected to the **E**, and we compare these data. If it is necessary to exchange the data, **E** transfers the data to **S**. **E** has a flag that signals whether data has been exchanged or not with **S**.

T is the raw data sent from **S** to be returned.

When **S**, **E**, and **T** finish exchange processing, **T** checks each **E** if exchange has been replaced with **chain_ch**. **E** accumulates flags indicating whether exchange has been replaced by received information from one of the **chain_ch**, and transfers to another **chain_ch**.

These flags are defined as zero when swapping occurs or one when there is no swapping. If these have been replaced at any one place, they continue to be able to process sorting.

T sends a message to **I** with **sync_ch** by continuing to determine sorting with **chain_ch**.

When **I** receives a message with **sync_ch**, it continues to sort for each **S**, or exits sorting and transfers data to each **S**. While continuing to sort, **S** repeats the process of sorting and exchanging. When sorting is completed, **S** returns the results to **I**.

I holds the final data which will be sorted by combining the data transferred from **S**.

6.2.2 Experimental Results

Table 3 Star model relations for process names, number of processes and channel-end.

Process names	Number of processes	Number of channel-ends
I	1	M + 1
S	M	3
E	M - 1	4
T	1	5

Table 3 lists the relations between processes. A Channel-end is the connecting point between a process and a channel. M is the number of **S** processes. This process involves sorting where that the process has a data set. This means the number of **S** and the number of split data sets.

This means that the number of channel-ends on **I** depends on the number of **S** processes. A process is implemented in a core. Fewer than 32 channel-ends could be implemented on one core. We need to take into account this limited number when implementing channel-ends.

We need to consider hardware constraints when implementing each process in each core. **I** and **T** were implemented in Core[0],

© 2012 Information Processing Society of Japan

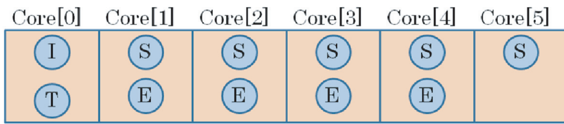


Fig. 7 Implementation arrangement for star model.

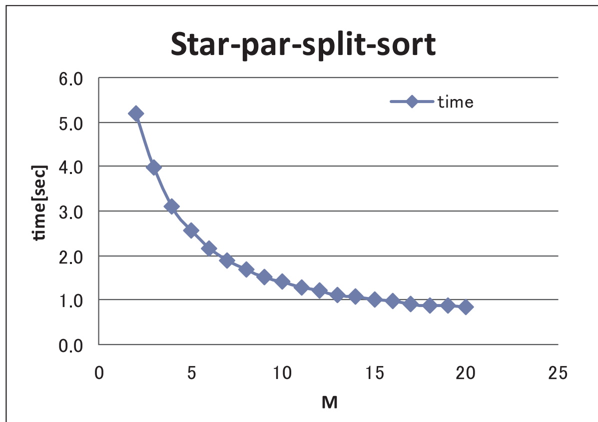


Fig. 8 Relations between running time and number of splits (M) with starred model.

and the other processes were implemented in Core[1] or higher.

Figure 7 shows when the number of splits equals five.

When M is more than six, **E** will be implemented in Core[5]. Then **S** and **E** will be implemented in order in Core[1].

Each core can implement a maximum of eight processes. One core can implement four sets (**I** and **T**) in this model. Process **S** in these cores can implement a maximum of eight processes. This shows it is possible to make twenty splits.

Figure 8 shows the relation between the number of splits M with running time. The M is the number of **S** processes. The time is the running time.

This shows that the running time asymptotically converges and we found that running time decreased with increasing numbers of splits in the measurement range.

6.3 Experiment on Our Ring Model

6.3.1 Design

We implemented **Ring-par-split-sort** for Parallel Split Sorting with ring model (Fig. 6 (Right)).

The ring model is composed of IO process (**I**), sorting (**S**). **I** was included in the ring model's **I** and **T**. **S** was included in the ring model's **S** and **E**. This model simplified the complex design. This is because it is difficult to understand the whole process by increasing the number of steps in the process and the frequency of communication.

A ring means that S and I are connected to a ring. Therefore, split_ch connected to I is constant regardless of the number of splits M.

This model's configuration is very similar to the flow of the algorithm. It is necessary to reduce the number of processes when this model is implemented, which also greatly reduces the number of channels. As the processes are connected in ring shaped, channel-ends are not concentrated in a single process. Therefore, it is affected by hardware limitations.

The star model process was reused in this experiment to implement the ring model. This was intended to reduce the effort spent in development.

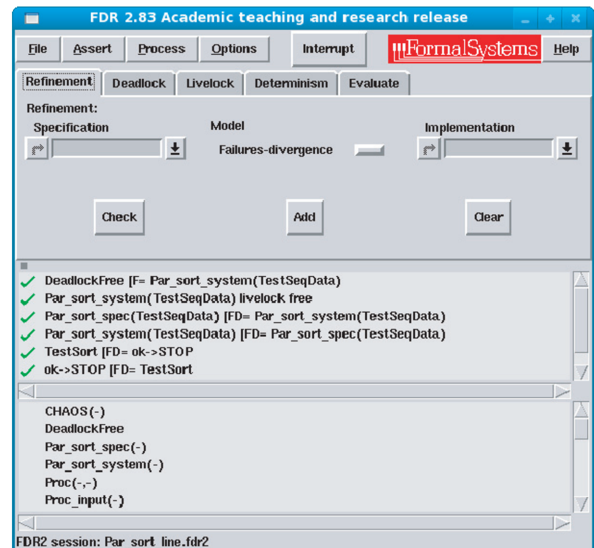


Fig. 9 FDR verification result for ringed split sorting algorithm.

6.3.2 Verification Tool

FDR [10] is a tool for automatically checking the CSP specifications for dead-lock freeness, live-lock freeness, and refinement relations. That is, FDR is a model-checking tool for CSP. FDR is also used to check refinement relations in failure divergence.

We specified a parallel split sorting algorithm using a CSP process. Processes communicated and collaborated with one another through a synchronized message-passing mechanism. We had to specify an appropriate order for messages to be sent and received for this reason. The final result was not correct or the system was deadlocked if this order was not correct.

We verified an equivalent relation of a refinement in failure divergence between CSP specifications for the parallel split sorting algorithm and a model for hiding internal events by using FDR. **Figure 9** shows a screenshot of the FDR checker.

Verification ensured that specifications behave correctly. Verification was used to determine the cause of failures and early resolution in this experiment.

6.3.3 Case Studies of Execution and Verification

Here, we explain how the process of our concurrent sorting is performed by giving case study.

- Case study I: Execution of concurrent sorting

  Figure 10 is a composition of Ring-par-split-sort.

  IO process is a process that transfers data to **Sort** processes, monitors the completion of the sorting, and collects the sorted data. Only one **IO** process is implemented.

  Sort process is a process that sorts the subset of data provided by the **IO**. The **Sort** process consists of M_MAX pieces, and has a unique ID within the range of 0–M_MAX-1.

  The **IO** and M_MAX **Sort** processes are connected to the ring, with channel **fw_chs** and **bk_chs** between processes. **Fw_chs** and **bk_chs** consist of M_MAX+1 pieces respectively, and both have the unique ID within the range of 0–M_MAX.

© 2012 Information Processing Society of Japan

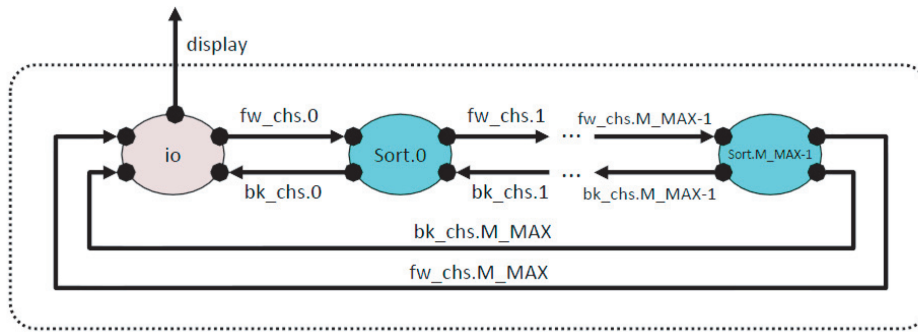


Fig. 10 Composition of Ring-par-split-sort.

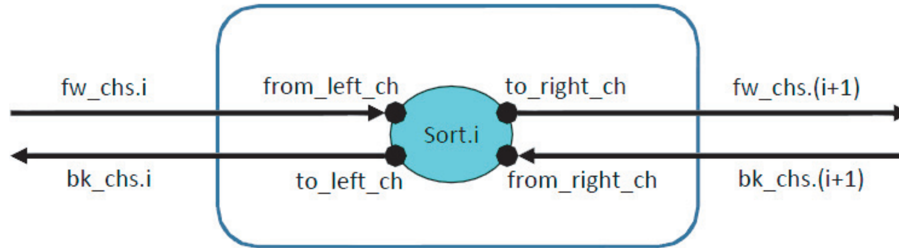


Fig. 11 Connection of channels and channel ends.

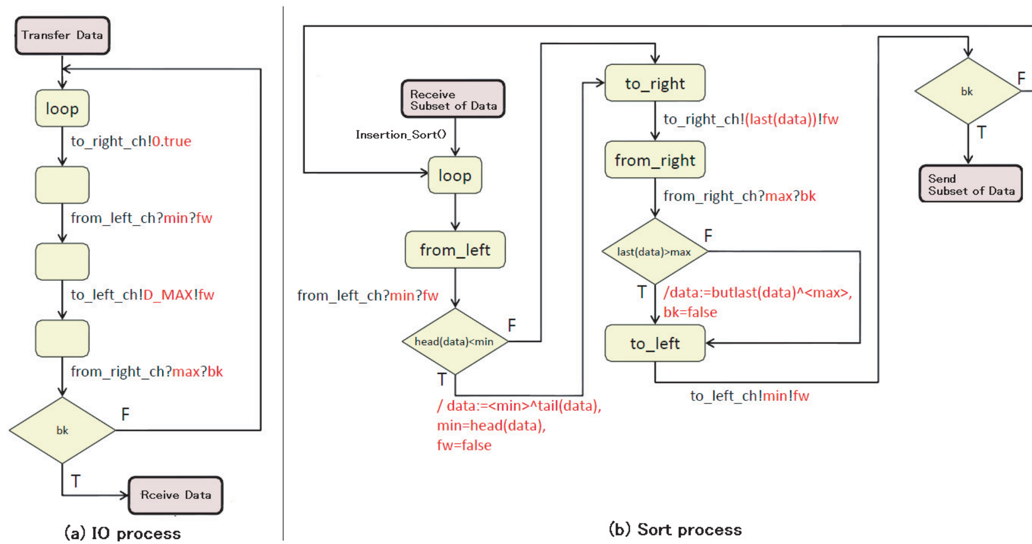


Fig. 12 Flow of Exchange Neighbors (with Ring-par-split-sort).

The original data set is divided into sub data, and each is distributed from the leftmost **Sort** process to the rightmost **Sort** process by the **IO** process.

Sorting is planned to line up the data over all processes from the leftmost process to the rightmost process, in ascending order.

Finally, the sorted data is returned to the **IO**.

Figure 11 shows the connection of channels and the channel ends of the processes. The **Sort** process is equipped with channel ends of **from_left_ch**, **to_right_ch**, **from_right_ch**, and **to_left_ch**. It is connected to the channel end of the adjacent processes through the channel **fw_chs** and **bk_chs** as shown in Fig. 11.

Ring-par-split-sort is an implementation of Parallel Sprit Sorting, as shown in Fig. 4 of Section 5.3. In each **Sort** process, it sorts the sub data set internally (Sorting in Fig. 4), then exchanges the data between adjacent processes (**Exchange Neighbors** in Fig. 4). The detailed processing flow in the part of **Exchange Neighbors** is shown in **Fig. 12**.

From_left_ch, **to_right_ch**, **from_right_ch**, and **to_left_ch** are channel ends to achieve the waiting state needed for communications. In the waiting state, a receive event or send event (ex: **from_left_ch?min?fw**) changes the state to the next.

Head(data) is the minimum value among data of the **Sort** process. **Min** is data just received from the **from_left_ch** channel end. The data is input from the left process, on the latest event.

Last(data) is the maximum value among data of the **Sort** process. **Max** is data just received from the **from_right** channel end. The data is input from the right process, on the latest event.

© 2012 Information Processing Society of Japan

Exchange Neighbors is an operation that performs the comparison and the exchange of **head(data)** and **min**, and **last(data)** and max, respectively.

Fw and **bk** are tokens. These tokens show that whether or not the sorting has been completed. Each token has the two states 'true' or 'false', and 'true' means to exchange freely (Fig. 12 (a)). **Fw** initializes with 'true' in **IO**, goes around **sort.0**, **sort.1** ... and **sort.M_MAX-1**, and then returns to **IO**. When **IO** receives the token **fw** from the **sort.M_MAX-1** process, **IO** sends it back to the **sort.M_MAX-1** process as **bk**. The token travels via **sort.M_MAX-1**, **sort.M_MAX-2**, ... and **sort.0**, and then returns to **IO** again. The states of tokens **fw** and **bk** become false when once there occur exchange among the **Sort** processes. It can be judged that sorting of all processes is completed if the states of **fw** and **bk** are true after travelling over all processes of the loop.

Next case study, we explain how the FDR model checker verifies by giving case study.

- Case study II: Verification with FDR

  FDR is Failure-Divergence Refinement model checker. With this refinement tool, a concurrent model description can be automatically validated from the model specification. All reachable states are checked, inspection whether error-free or not is performed, and non-deterministic features; namely livelocks or deadlocks are detected. When a concurrent model and the specification are verified to be equivalent by failure divergence, the concurrent model satisfies the specification. FDR performs failure divergence among models, so it can evaluate the equivalence of concurrent implementations and the specifications. The Ring-par-split-sort model is verified by using FDR.

  Figure 13 shows a part of the specification description to our concurrent model; deadlock-free and livelock-free are verified in the model.

  In Fig. 9, the result shows that the verification process is completed in the normal terminations (checks on green), and the abovementioned features are verified without failures.

  Therefore, our Ring-par-split-sort has been proven to be deadlock-free, and livelock-free.

6.3.4 Results from Experiment

Table 4 lists the relations between processes. A channel-end is a connecting point between a process and a channel. M is the number of **S** processes. This process involves sorting processes that have data sets. This means the number of **S** and the number of split data sets.

This means that the number of channel-ends on **I** does not depend on the number of **S** processes.

We need to consider hardware constraints in implementing each process in each core. **I** was implemented in Core[0]. **S** were implemented in Core[1] or higher.

Figure 14 shows the number of splits when M is five. When M is six or more, more than one S is appropriated in Core[1] or higher.

Since each Core[1-5] appropriates a maximum of eight processes, it is possible to make forty splits.

Figure 15 plots the relation between the numbers of splits M with running time. The M is the number of **S** processes. The time is running time. This indicates that the running time decreases up to M = 22. However, when M is more than 22, the running time slowly increases.

```
--------------------------------------------------------
--              Verification
--------------------------------------------------------
-- The system is deadlock free
DeadlockFree=( |~| x:Events @x -> DeadlockFree) |~| SKIP
assert DeadlockFree [F= Par_sort_system(TestSeqData)

-- no livelock
assert Par_sort_system(TestSeqData) :[livelock free [FD]]

-- the correct result can be displayed.
assert Par_sort_spec(TestSeqData) [FD=Par_sort_system(TestSeqData)
assert Par_sort_system(TestSeqData) [FD= Par_sort_spec(TestSeqData)

--------------------------------------------------------
--           check isort_list
--------------------------------------------------------

channel ok,error

chksort(<>)= true
chksort(<x>)= true
chksort(<x,y>^s) = if (x <= y) then chksort(<y>^s) else false

TestSort
= [] s:SeqDataSet @
 if chksort(isort_list(s))
 then
  ok -> STOP
 else
  error -> STOP

assert TestSort [FD= ok -> STOP
assert ok -> STOP [FD= TestSort
```

Fig. 13 FDR description.

Table 4 Ring model relations for process names, number of processes and channel-end.

Process names	Number of processes	Number of channel-ends
I	1	2
S	M	2

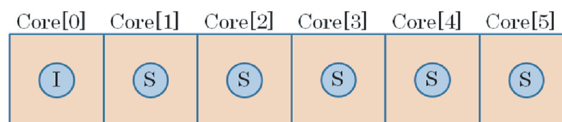


Fig. 14 Implementation arrangement for ring model.

6.3.5 Evaluation of Performance

We found from these experimental results that the execution time could be expected to be reduced by increasing the number of divisions. However, the results indicated that the execution time increased as a boundary point with too many divisions.

In **Fig. 16**, we show the ratio of sorting time and neighbor exchanges (communication time) in the execution time of one cycle for Sorting and neighbor exchanges on increasing M.

Execution time (E) is determined by R_M times the sum of sorting time (s) and communication time (x).

R_M is the number of repeat cycle for Sorting (in Fig. 4) and Exchange Neighbors (Fig. 4) between sorting start and finish.

© 2012 Information Processing Society of Japan

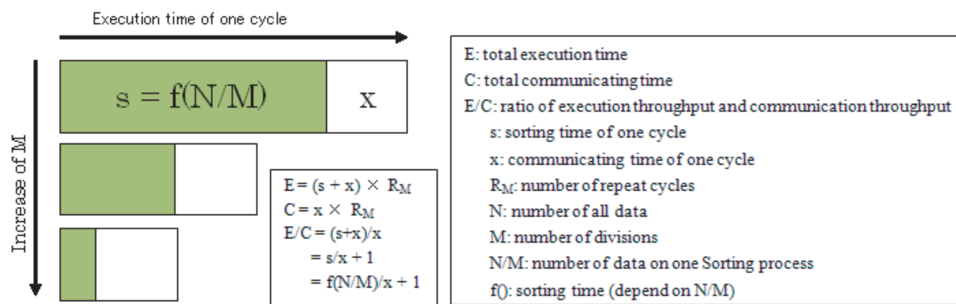


Fig. 16 The ratio of sorting time on increasing of M.

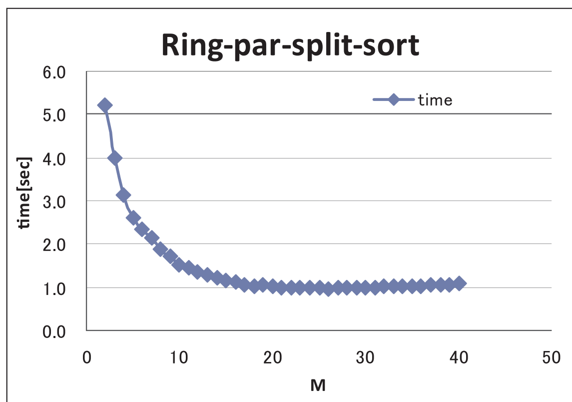


Fig. 15 Relations between running time and number of splits (M) with ring model.

Table 5 Execution time of non-parallel insertion sort and Ring-par-split-sort.

Num of Sort process	Execution time(N=10000)[msec]
non-parallel	10583
2	12697
4	7434
8	4118
10	3676
12	2783
18	2099
20	1799

M is the number of divisions. The repeat count is $R_M < R_{(M+1)}$.

Sorting time (s) is determined by a function of the number of data (N) and M depended.

For example, the computing time for insertion sort is $f((N/M)) > f(N/(M + 1))$.

Neighbor exchange involves comparing and exchanging being repeated once across each node.

The communication time of one cycle (x) becomes constant, and it is independent of M.

The total execution time (E) and the total communication time (C) are defined by the expressions in the square of Fig. 16.

From the left of Fig. 16 makes it clear that there is a rise in the percentage of the communication time in the execution time with increasing M. E/C is the ratio of execution throughput and communication throughput, and it is inversely proportional to M. When M is small, the change in the execution time by decreasing s is large. This change is the main benefit of parallelization. The percentage of communication in the execution time will increase by increasing M.

6.3.6 Effectiveness Comparison of Concurrency

Our parallel split sorting algorithm has the $O(N^2)$ time and $O(N)$ space computational complexity. We show the execution time of the non-parallel insertion sort and the Ring-par-split-sort in **Table 5**.

6.4 Discussions

In this experiment, parallelization of sorting was implemented on the CSP based methodology. We designed and implemented a system with two concurrent sorting models by focusing on the requirements for resource constraints and hierarchy.

[Concurrency]

We found from the experimental results that the execution time could be expected to be reduced by increasing the number of divisions. However, the results revealed that the execution time increases as a boundary point with too many divisions.

[Hierarchy]

CSP based architecture enables cooperation with synchronous communications, making it easy to develop it into hierarchal structures among module units. The modules in the star model were easy to reuse in developing the ring model. Our experiment revealed that it was able to reduce the number of development processes.

[Resource Constraints]

Embedded systems have resource constraints. Our experiment revealed we could reduce the number of processes while retaining concurrency. We found resource usage could be reduced by comparing the star model and the ring model.

[Safety]

Inter-process communications tend to become complicated in parallelizing sequential processes. This problem was resolved with the CSP based architecture, because modules were able to determine the cause of bugs and to ensure the safety of specifications. Our experiments revealed FDR is a useful tool for verifying the safety of software modules.

We implemented a sorting algorithm on practical hardware, which contained the basic mechanisms for controlling data. We were able to apply the sorting algorithm in embedded systems easily by replacing these data control mechanisms with device control instructions. The CSP based architecture satisfied the requirements of having a hierarchical structure, resource con-

© 2012 Information Processing Society of Japan

straints, and safety, as previously mentioned. Therefore, CSP based architecture can be said to be useful in the development of embedded concurrent systems. Moreover, the sorting algorithm is good benchmark software to test the properties of embedded systems.

7. Concluding Remarks

In this paper, we proposed a model of parallel sorting and implemented it in an experiment. We tried to measure results by running it on hardware and we evaluated the performance of concurrency by focusing on the ratio of communication time in execution time. We also discussed the usefulness of the CSP-based architecture. Although there are several approaches to parallelization in sequential processing, CSP can be developed to meet the requirements of hierarchy and safety.

We are considering future experiments on evaluating power-savings in embedded systems.

Acknowledgments The authors would like to express their thanks to Prof. Zenjiro Ohba of Toyo Univ. and Dr. Kazuhito Matsui of CSP Consortium for helpful discussions.

References

[1] Hoare, C.A.R.: Communicating sequential processes, *Comm. ACM*, Vol.21, No.8, pp.666–667 (1978).
[2] Roscoe, A.W.: The theory and practice of concurrency, Prentice Hall (1998).
[3] Zurawski, R.: *EMBEDDED SYSTEMS DESIGN AND VERIFICATION*, CRC Press (2009).
[4] Narayan, S.: Requirements for Specification of Embedded Systems, *Proc. 9th Annual IEEE International ASIC Conference and Exhibit*, pp.133–137 (1996).
[5] Saifhashemi, A. and Beerel, P.A.: High Level Modeling of Channel-Based Asynchronous Circuits Using Verilog, *Communicating Process Architectures 2005*, pp.275–288, IOS Press (2005).
[6] Mori, H.: Study of Embedded Concurrent Systems, The Institute of Electronics Information and Communication Engineers (IEICE) FIIS-11-307 (in Japanese), pp.1–7 (2011).
[7] Nakajima, S.: Formal Methods as Software Engineering Tools, National Institute of Informatics NII-2007-007J (2007) (in Japanese).
[8] XMOS Limited, available from ⟨http://www.xmos.com/⟩.
[9] Breshears, C.: *The Art of Concurrency*, O'REILY (2009).
[10] Formal Systems (Europe) Limited, available from ⟨http://www.fsel.com/⟩.
[11] Hasegawa, Y., Isobe, Y., Ohmaki, K., Mori, H. and Tsuchida, K.: Parallel Processing by CSP based Architecture, The Institute of Electronics Information and Communication Engineers (IEICE) FIIS-10-278 (in Japanese), pp.1–9 (2010).
[12] Thanakulwarapas, T. and Werapun, J.: Communication-Space Efficient Parallel Bitonic Sorting on Symmetric Multiprocessors, *Advances in Computer Science and Technology* (*ACST 2008*), pp.75–79 (2008).
[13] Lee, J.D. and Batcher, K.E.: Minimizing Communication in the Bitonic Sort, *IEEE Trans. Parallel and Distributed Systems*, pp.459–473 (2000).

Yuki Hasegawa received his M.E. degrees in Open Information Systems from Toyo University in 2011. His research interests include parallel and concurrent architectures.

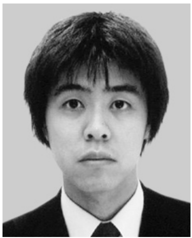
Yoshinao Isobe received his B.E. and M.E. degrees in Electrical Engineering from Shibaura Institute of Technology in 1990 and 1992 respectively. In 1992, he joined Electrotechnical Laboratory, MITI. He received his D.E. degree from Shizuoka University in 2001. He was a visiting researcher of the University of Wales, Swansea for one year in 2003. He is currently a senior researcher in the National Institute of Advanced Industrial Science and Technology. His research interests include formal verification of concurrent systems. He is a member of IEICE, JSSST, and IPSJ.

Kazuhito Ohmaki received his M.S. and Ph.D. degrees in Computer Science from Tohoku University in 1976 and 1979 respectively. He joined Electrotechnical Laboratory, Agency of Industrial Science and Technology in 1979. He was an invited researcher af ETH Zurich in 1984. He has been a professor of Faculty of Information Sciences and Arts at Toyo University since 2009. His major interests include Software Engineering based on formal semantics. He is a member of IPSJ, JSSST, ACM and IEEE.

Hideki Mori received his M.S. and Ph.D. both from Keio University in 1974 and 1978, respectively. He joined the Faculty at Toyo University from 1978 to 2008. He was a Visiting Scholar of the Department of Computer Science of UCLA in 1984. Currently, he is a visiting professor of the Graduate School of Open Information Systems of Toyo University, and a Visiting Professor of Tokyo Denki University. His research interests include parallel and concurrent architectures, and fault-tolerant computation. Most recently, his research has emphasized on concurrent systems. He is a member of IEEE, ACM, IPSJ and IEICE.

© 2012 Information Processing Society of Japan

Kensei Tsuchida received his M.S. and D.S. degrees in mathematics from Waseda University in 1984 and 1994 respectively. He was a member of the Software Engineering Development Laboratory, NEC Corporation in 1984–1990. From 1990 to 1992, he was a research associate of the Department of Industrial Engineering and Management at Kanagawa University. In 1992 he joined Toyo University, where he was an instructor until 1995 and an associate professor from 1995 to 2002, and a professor of from 2002 to 2009 at the Department of Information and Computer Sciences and since 2009 he has been a professor of Faculty of Information Sciences and Arts. He was a visiting associate professor of the Department of Computer Science at Oregon State University from 1997 to 1998. His research interests include software visualization, human interface, graph languages, and graph algorithms. He is a member of IPSJ, IEICE Japan and IEEE Computer Society.



Yasunori Shiono received his M.E. and D.E. degrees from Toyo University in 2006 and 2010 respectively. He is currently an assistant professor of Faculty of Information Sciences and Arts at Toyo University. His research interests include graph algorithms, graph grammars, fuzzy theory and software development environments. He is a member of IEICE Japan, JSSST, JSIAM and IEEE.

© 2012 Information Processing Society of Japan