Reducing Overhead in Implementing
Fine-grain Parallel Data-structures of a Dataflow Language
on Off-the-shelf Distributed-memory Parallel Computers

Shigeru Kusakabe, Taku Nagai, Ken taro Inenaga, and Makoto Amamiya
Kyushu University, Japan
E-mail: \{kusakabe, taku, inenaga, amamiy\}@is.kyushu-u.ac.jp

Abstract

In order to show the feasibility of a fine-grain dataflow computation scheme, we are implementing a fine-grain dataflow language on off-the-shelf computers, using a fine-grain multithread approach. Fine-grain parallel data-structures such as I-structures provide high level abstraction to easily write programs with potentially high parallelism. The results of preliminary experiments on a distributed memory parallel machine indicate that the performance inefficiency related to fine-grain parallel data-structures in the naive implementation is mainly caused by the calculation of the local address for distributed data, and the frequent fine-grain data access using message passing. In order to reduce the addressing overhead, we introduce a two-level table addressing technique. We employ a caching mechanism and a grouping mechanism for the fine-grain data access. The preliminary performance evaluation results indicate that these techniques are effective to improve the performance.

1. Introduction

Parallel programming often requires troublesome execution control such as explicit scheduling and synchronization of subcomputations. Specifying control of parallel execution is extremely hard especially in a procedural approach where control must be explicitly addressed. In a dataflow approach, on the other hand, timing control of parallel execution is abstracted away because synchronizations are resolved automatically along with data dependencies. By using a dataflow language with fine-grain parallel data-structures such as I-structures[2], we can easily write highly parallel programs in such a way that complex parallel execution control for data-structure handling is abstracted away.

However, efficient implementation platform for fine-grain dataflow languages with fine-grain parallel data-structures used to be limited to special architectures[1, 7]. In order to show the feasibility of a fine-grain dataflow language, we are implementing the fine-grain dataflow language[10] on off-the-shelf computers, using a multithread computation approach. In a naive implementation, we generate very fine-grained codes, in which a lot of dependencies are resolved at runtime, in order to ensure non-strictness of the language. But such naive fine-grained codes run inefficiently on machines which do not provide hardware support for the fine-grain execution. As far as arrays of I-structure are concerned, the results of preliminary experiments on a distributed memory stock parallel machine Fujitsu AP1000[13] indicate that the performance inefficiency in the naive implementation is mainly caused by the calculation of the local address for distributed data, and the frequent fine-grain data access using message passing. In this paper, we discuss techniques to reduce the overhead in implementing fine-grain parallel data-structures on stock parallel machines.

Our implementation employs a multithread execution model in which data-structures are not token data but heap memory objects. Array access is performed by means of pointers to heap areas. Thus, many optimization techniques proposed for conventional languages are also applicable to our implementation of fine-grain parallel data-structures, although our language is a fine-grain non-strict dataflow language. In order reduce the overhead caused by the runtime address calculation, we introduce a two-level table addressing technique for access sequences to the array of fine-grain parallel data-structures. In order to reduce the overhead caused by the frequent fine-grain data access, we employ a caching mechanism and a grouping mechanism for the fine-grain data access. During the
optimization process, following issues are considered. Fine-grain parallel data-structures of our language allow element-by-element access. This feature is one of the key features to make non-strict programs as well as parallel programs. Optimizations involving discreet grouping may reduce effective parallelism, and may lead to a deadlock at runtime in the worst case.

In section 2, we describe our fine-grain parallel execution model on stock machines. In section 3, we show our implementation model of fine-grain parallel data-structures. In section 4, we point out problems to be solved in order to efficiently implement fine-grain parallel data-structure. In section 5, we present a two-level table addressing technique that can efficiently generate the address of the array elements of fine-grain parallel data-structures. In section 6, we discuss a fine-grain data caching mechanism. In section 7 and 8, we discuss a grouping mechanism for the fine-grain data access. The preliminary performance evaluation results indicate that these techniques are effective to efficiently implement fine-grain parallel data-structures on stock parallel machines.

2. Fine-grain execution on stock machines

Our runtime model on stock machines adopts a multithread execution model. Since commodity processors exploit locality of computation to achieve high performance, direct mapping of the dataflow-based fine grain execution to such machines is inefficient. We adopt the following runtime model that can utilize computational locality as well as fine grain parallelism. In this paper, a thread refers to an instruction sequence to be executed exclusively. The runtime model is latency tolerant, and is expected to achieve high throughput by exploiting pipeline parallelism between producers and consumers.

Programs of our language consists of number of functions, and function instance level parallelism is exploited by using a data structure, called frame, which is allocated to each function instance. Intra-instance parallelism is exploited as multithread parallel processing. Figure 1 shows a schematic view of a frame. The variable slots area is used to manipulate local data. The thread stack is provided in order to dynamically schedule intra-instance fine-grain threads. When a thread becomes ready to run, the entry address of the thread is pushed onto the stack. A processor executes the instructions of a thread in a serial order until the termination point of the thread. If the execution of the current thread terminates, the next ready thread is popped from the top of the thread stack for the next execution. The processor repeats this operation until the the ready threads are exhausted, then switches to another frame. Local variables are not token data, but mapped to variable slots on the frame. Arrays are memory objects on heap areas and manipulated by means of pointers to the heap areas. Thus, many optimization techniques for conventional languages can be introduced in our implementation.

3. Fine-grain parallel data-structures

3.1. Making an array: mkarray

An array is generated by a bulk operator, mkarray. In creating a non-strict array, the allocation of the array block and the computations to fill the elements are separated. In a dataflow computation scheme, a mkarray immediately returns an array block (pointer) and calls the filling functions in parallel. Figure 2 shows a schematic view of a mkarray computation model. In the figure, the mkarray node is an instance representing an array, and pfs are instances of filling functions for array elements. Although the pfs are activated in parallel, they may be serialized according to data dependencies.

3.2. Making an array on distributed parallel machines

On distributed-memory machines, arrays are often distributed over the processor nodes, and filling functions are scattered among the processor nodes. We map the filling functions using the owner computes rule as a basic rule. Figure 3 outlines the mkarray computation model on a distributed memory machine.

Array descriptors support array distribution and realize global address access for distributed arrays. Array
Figure 2. Schematic view of a mkarray computation model

Figure 3. Schematic view of a mkarray computation model on a distributed memory machine.

Descriptors consist of a body, a decomposition table, and a mapping table, in order to manage the top-level information, the decomposition scheme, and the mapping scheme, respectively. The body has a total size, the number of dimensions, the number of processors, and various flags. The decomposition table has the lower and the upper bounds of each dimension, and some parameters to specify the decomposition scheme: the number of logical processors, the block size, and the baseline. The mapping table has correspondences between the logical and physical processors for the subarrays, the top of the local address for the block, and a caching area.

Figure 5 illustrates a read request across processors:

1. The thread that has a read request on the processor X resolves the target address (processor ID and memory offset in the target processor memory). The read request is sent to the target processor (in this case, processor Y) together with the continuation thread in the requester instance. Within the processor X, when the thread that issued the read request terminates, other ready threads are scheduled until the read request is completed.

2. When the read message is accepted at the processor Y, a message handler manages the read request. If the target element data is empty, the read request is deferred and the continuation in the read request is queued in the pending list.

3. When the element data is stored, the data is returned to the continuation thread within the requester instance on the processor X according to the continuation information in the pending list.

4. In the acceptance of the returned data message at processor X, the continuation thread becomes ready and will be scheduled.

Figure 4. Array descriptor

Figure 5. A read request across processors
4. Overhead of fine-grain parallel data-structures

In order to confirm the cause of the performance inefficiency of fine-grain parallel data-structures, we examined the results of preliminary experiments on the early implementation. As an example, we used a CYK (Coche-Younger-Kasami) parsing program[8, 15]. Fine-grain implicit parallel data-structures, such as I-structures are effective in writing an elegant CYK program for parallel processing. The CYK program performs an irregular array access pattern during the program execution. Figure 6 is a profile of the CYK program on the AP1000. This CYK generates an \( n \times n \) array for an \( n \)-symbol input. In this experiment, the array are distributed as \( (p = 16; k = 1) \) in the column, and \( (p = 1; k = n) \) in the row, where \( p \) is the number of processors, \( k \) the block size. We employ the owner computes rule as a basic distribution scheme.

![Figure 6. Profile of a CYK parsing program on the AP1000](image)

In the figure, the dark-shaded part shows the estimated time of the actual computation. Overhead is much larger than the actual computation. The overhead is classified as:

- **LIB**: AP1000 OS time. This includes overhead of message send/receive operations.
- **THREAD SW**: Thread switching overhead.
- **FRAME SW**: Frame switching overhead.
- **SYNC, PEND & RESUME**: Overhead due to fine-grain synchronization management such as synchronizing access to I-structures.
- **etc**: Miscellaneous software overhead such as message pack/unpack overhead and execution profiling overhead.
- **ADDRESS**: "Subscript \( \rightarrow \) physical address" calculation overhead at runtime.

From this result, we conclude that the main causes of the overhead in implementing fine-grain parallel data-structures are, in addition to multithread execution management such as thread switching, (1) the runtime calculation of the local address for distributed data, and (2) the frequent fine-grain data access using message passing.

In order to reduce overhead caused by (1), we introduce a two-level table addressing technique for the access sequence to fine grain parallel data-structures. In order to reduce overhead caused by (2), we employ a caching mechanism and a grouping mechanism for the fine-grain data access.

5. Addressing optimization

The results of preliminary experiments using I-structures arrays indicate that one half of the performance inefficiency in the naive implementation is caused by the local address calculation for distributed data. Although our language is a fine-grain non-strict dataflow language, many optimization techniques proposed for conventional languages are also applicable to our implementation of fine-grain parallel data-structures without affecting the non-strictness.

In order to reduce the overhead caused by the runtime address calculation, we introduce a two-level table addressing technique for access sequences to the array of fine-grain parallel data-structures.

5.1. Transitive addressing

It is necessary to specify the real address of the memory, although we use a subscript to specify an array element at the source program level. While the calculation of the real address can be performed only at runtime if static analysis fails, some transitive addressing methods to reduce addressing overhead are applicable if an array access sequence has some kind of regularity. Transitive addressing methods generate information useful to avoid repeating naive runtime address generation for array access sequences. (See figure 7.) Although such methods were originally proposed for procedural languages they can be used for implementing dataflow languages as well.

5.2. Table-driven approach

The method, which our addressing optimization is based on, was developed for array access sequences.
with a fixed stride called a regular section[4]. This method generates, and holds in a table, essential information to determine the local address of the elements that are allocated and used within a processor. At runtime (compile time if possible), addresses are generated transitively by using the information held in the table.

The main point of this table addressing is that, under the assumption of block-cyclic(k) distribution, a local memory access sequence on any processor represented by a regular section (l : h : s), can be characterized by a finite state machine of at most k states. (Where: k is the block size, l the lower bound, h the upper bound, and s the stride. We also use p as the number of processors, and b as the alignment offset in the rest of the paper.)

For example, consider a one-dimensional array \( A[0..\text{large}] \) distributed by \( p = 3, k = 4 \) and \( b = 0 \), and a regular section \( A(0: \text{large}: 5) \). Figure 8 shows an array \( A \) distributed over logical processors and a regular section \( A(0: \text{large}: 5) \) (boxed elements in the figure).

\[
\begin{array}{cccc}
\text{PE 0} & \text{PE 1} & \text{PE 2} \\
0 & 1 & 2 & 3 \\
1 & 12 & 13 & 14 & 15 \\
2 & 24 & 25 & 26 & 27 \\
3 & 36 & 37 & 38 & 39 \\
4 & 48 & 49 & 50 & 51 \\
\end{array}
\]

Figure 8. An array \( A \) and a regular section \( A(0: \text{large}: 5) \)

Figure 9 shows the state table and the state transitions. \( \Delta C \) is the difference in the course of blocks within the processor holding the array element, and \( \Delta O \) the difference in the offset of the array element within course. \( \Delta M \) is the local memory offset, which is calculated by \( k \times \Delta C + \Delta O \). In this example, \( \Delta M \) is calculated as \( [7, 2, 0, 2] \).

5.3. Addressing of read operation

We applied the table addressing method to the CYK program. However, the number of improved access operations is only 314 out of 45088 (0.74 %). Although the table addressing method is suitable to store operations for arrays in parallel loops under the owner computes rule, the effect on the CYK program in our implementation was limited. The difference of execution model results in the limited effect. The table addressing is originally proposed for a different execution model from our model. They adopted Koelbel’s execution model for parallel loops[9], in which a processor goes through the following steps:

1. Send those data items it holds that are required by other processors.
2. Perform local iterations, i.e., iterations for which it holds all data.
3. Receive data items from other processors.
4. Perform nonlocal iterations.

On the other hand, in our model, read access requests are asynchronously issued to the owner processor. (See Figure 10.)

Since consumer processes are responsible for addressing in our model, we need to optimize read operations in our implementation. However, differently from store operations, it is necessary to generate the processor number (ID) as well as the local address for each array element. A read operation may require remote memory access even if we assume the owner computes rule. A key is that we can generate the local address if we can specify the processor number for the next element of the regular section of read operations. The \( \Delta P \), the offset between the processor ID for the current element and the processor ID for the next element, is also characterized by a finite state machine.
5.4. Evaluation of addressing optimization

Table 1 shows the estimated runtime overhead due to address generation. The overhead caused by naive runtime addressing, presented as “ss_address” in the table, is influenced by the number of the target array dimensions. “Increment,” “constant,” “table,” and “table + ΔP” in the table are transitive addressing cases. The most efficient case “increment” occurs when the stride of the regular section is 1. Next one, “constant” is the same as “increment” except that its stride is constant but not 1. Presented as “table” is the standard table addressing, and “table + ΔP” is the two-level addressing for regular sections of read operations.

By using the extended two-level table addressing technique, the performance of the CYK program is significantly improved. The number of improved access operations is 42227 out of 45988 (91.8%). Figure 12 shows the execution profiles of the naive code and the optimized code for the CYK program.

6. Fine-grain data caching

On distributed memory parallel computers, communication overhead may degrade the program execution performance.

In order to reduce inter-processor communication overhead, we introduce a fine-grain caching mechanism which buffers the fetched data into the caching area in the local memory of the consumer process. The consumer process needs a dynamic check to see if the target data item is buffered or not, since it is difficult
to statically determine the timing of access operations to an element of fine-grain data-structures. However, this caching mechanism need not to involve any complex consistency management mechanisms, since consistency management for cached data is not required, thanks to write-once semantics of our language.

We applied this technique to the CYK program and a matrix multiplication program. Figure 13 shows the result of the CYK program and Figure 14 the result of the matrix multiplication program. Vertical axis shows the elapsed time and horizontal the number of processors used.

![Figure 13. Effect of optimization for CYK](image)

While the performance for the matrix multiplication program improved significantly, the performance improvement of the CYK program was very little. This is because, under this caching mechanism, every data access operation involves a check whether the data is cached or not. The effectiveness of the mechanism depends on the number of access operations to a single data element. If there are not so many access operations to the identical data, the checking overhead may not be canceled. The array access pattern of the CYK program is irregular, and each consumer process does not perform many access operations to the identical data.

![Figure 14. Effect of optimizations for matrix multiplication](image)

7. Grouping fine-grain data

Ideally, fine-grain computation and communication are pipelined and overlapped in a dataflow computation scheme, without exposing the communication overhead. However, the communication overhead may degrade the performance, since the communication overhead is very high. Many communication optimizations proposed for procedural languages are also applicable to dataflow languages. There are some transformations to combine multiple messages and send them in a single operation:

**Message vectorization:** If the compiler can find the set of data items transferred in a loop, the compiler can group them and send them in a single block transfer rather than sending each element data in an individual message[16].

**Message coalescing:** The compiler can reduce the frequency of communication by grouping messages together that send overlapping or adjacent data[14].

**Message aggregation:** The compiler aggregate messages being sent to the same processor even if the data they contain is unrelated[14].

An element of A-structures has a single synchronization tag for the entire (or grouped) data object[6].
Figure 15.) Synchronization is performed not for each element data like I-structures, but for the entire (or grouped) data. The grain size of A-structures access is larger than that of I-structures access. A-structures access can be implemented using efficient block transfer.

![Diagram of state transitions of I-structures and A-structures]

**Figure 15. State transitions of I-structures and A-structures**

We examined the potential effectiveness of A-structures. We used a matrix multiplication as a safe program. Figure 16 shows the experimental result of $32 \times 32$ matrix multiplication program. As shown in the figure, the use of A-structures can provide potential performance improvement.

However, the problem is when and how to use A-structures effectively and safely. We have to give careful consideration to the following points:

- Serialization by using A-structures may reduce effective parallelism.
- Indiscreet grouping in generating an array object which has circular self-reference may lead to a deadlock at runtime.

Many non-strict dataflow programs have data dependencies to be resolved at runtime. It is difficult to "safely" apply the technique listed above to such programs. For instance, the CYK program uses non-strict data structure, and indiscreet grouping may lead to a deadlock.

However, some techniques to grouping messages are worth introducing even into our implementation of the dataflow language.

8. **Message coalescing**

Although techniques to group messages are worth introducing and a static A-structures approach would be effective, the problem is when and how to introduce such an approach effectively and safely. Even if grouping messages does not lead to a deadlock at runtime, compile-time serialization may reduce effective parallelism at runtime.

We are developing a coalescing mechanism for manipulating fine-grain parallel data-structures. The coalescing mechanism can reduce the frequency of communication by grouping messages for adjacent data of fine-grain parallel data-structure without losing the non-strictness. Figure 17 shows a schematic view of the mechanism:

- Read requests to adjacent data on a remote producer (owner) from a consumer process are packed at compile time. In Figure 17, requests to $V[i, j]$, $V[i-1, j]$, $V[i+1, j]$, and $V[i,j-1]$ on PE1 are packed and sent to PE1 in a single message.
- When the producer replies, all available data items are packed in a single message and replied to the consumer. In Figure 17, available data, $V[i, j]$, $V[i-1, j]$, and $V[i+1, j]$ are packed in a single message and returned to the consumer.
- Requests to unavailable data elements in the producer are suspended as deferred requests of I-structures. In Figure 17, $V[i,j-1]$ is unavailable, and the request is suspended.

As a test program for our coalescing mechanism, we use an edge detection program of GNC (Graduated Non-Convexity) algorithm[3]. Each element of an $n \times n$ array repeats updating its own value using the
8 neighbors’ value. This program is a data-parallel program, but there is an irregularity at the edge of distributed data block on distributed memory parallel machines. Introducing A-structures is difficult and serialization by indiscriminate grouping may reduce effective parallelism.

Table 2. Effect of optimizations for the GNC program.

<table>
<thead>
<tr>
<th></th>
<th>4x4 PE (256 items/PE)</th>
<th>8x8 PE (64 items/PE)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Speedup (%)</td>
</tr>
<tr>
<td>naive</td>
<td>5.61</td>
<td>—</td>
</tr>
<tr>
<td>addressing</td>
<td>4.28</td>
<td>15.5</td>
</tr>
<tr>
<td>coalescing</td>
<td>4.21</td>
<td>16.4</td>
</tr>
<tr>
<td>addressing +</td>
<td>4.13</td>
<td>18.5</td>
</tr>
<tr>
<td>coalescing +</td>
<td>4.72</td>
<td>8.9</td>
</tr>
<tr>
<td>caching</td>
<td>4.26</td>
<td>16.0</td>
</tr>
</tbody>
</table>

In addition to the coalescing, we also examined the caching mechanism for the GNC program. The estimated times of communication and tag checking for the coalescing version and the caching version are almost the same. While message length of the coalescing version tends to be longer than that of the caching version, message length does not have serious impact on the performance of the API1000. But the coalescing is more effective than the caching in this case. This is because the caching version causes more context switching than the coalescing version. As seen from this result, this grouping messages mechanism is effective in our implementation of the dataflow language.

9 Concluding Remarks

We discussed implementation issues of fine-grain parallel data-structures on stock parallel machine. We introduced a two-level table addressing in order to reduce the addressing overhead for a regular section of read operations. We also examined a caching and a grouping mechanisms, which do not violate the nonstrictness of the language, for the fine-grain data access. The preliminary performance evaluation results indicate that these techniques are effective to improve the performance. We believe that adequate techniques make feasible such fine-grain parallel data-structures even on stock distributed parallel machines which do not provide special support for such fine-grain data-structures.

References


