Reducing Overheads of Local Communications in Fine-grain Parallel Computation

Jin-Soo Kim   Soonhoi Ha   Chu Shik Jhon

Department of Computer Engineering
Seoul National University
Seoul 151-742, KOREA
{jinsoo, sha, csjhon}@comp.snu.ac.kr

Abstract

For fine-grain computation to be effective, the cost of communications between the large number of subtasks should be minimized. In this paper, we present an optimization technique which reduces overheads of communications between local subtasks by bypassing the network interface and transferring data directly from memory or registers to memory. On average, the optimization results in 35.6% improvement in total execution time on instruction-level simulations with six benchmark programs from 1 to 32 nodes.

1 Introduction

Fine-grain parallel computation has several advantages such as architecture independence, potential for exploiting parallelism, ease of use as a target for code generation, and capabilities of balancing loads and hiding communication latencies. An architecture that can exploit the fine-grain parallelism is a *multithreaded architecture*.

In parallel computation, communication latency between subtasks on different nodes is inevitable and keeping a certain number of subtasks in each node is necessary for multithreading to be useful. However, an operation that communication is expected at compile time may not generate any message at run time depending on the location of the target subtask. Those operations can not be determined statically because subtasks are created and destroyed dynamically and the number of subtasks itself is data-dependent. Moreover, the exact location of a subtask can not be predicted at compile time.

The proposed approach is to generate alternative codes at compile time that transfer data between local subtasks bypassing the network interface. Before generating a message, the program decides which code to execute according to the location of the destination. No message is generated in case of local communications and unnecessary context switching can be avoided.

2 A Model for Fine-grain Parallel Computation

In this paper, we are interested in a model of fine-grain multithreading, which can hide a long latency using a large number of threads. Also it is desirable for such model to be implementable using commodity microprocessors to exploit their high performance/price ratio. For these reasons, we have adopted the TAM (Threaded Abstract Machine) [1], a compiler-controlled multithreading model.

A TAM program consists of a collection of *code blocks*. Each code block represents a subtask, and typically specifies a function or a loop body. A code block is compiled into a set of *inlets* and *threads*. Inlets are short message handlers and threads are sequences of instructions that can not suspend. When a code block is invoked, a *frame* is allocated for storage of arguments, local variables, and a list of ready threads associated with the frame.

The TAM scheduling hierarchy consists of a two-level structure comprising a collection of frames, each containing one or more addresses of enabled threads in a region of the frame called the *remote continuation vector* (RCV). When a frame is activated, the list of ready threads in the RCV is copied into a special region called the *local continuation vector* (LCV). Threads are fetched and executed from the LCV until none remains, after which frame switching is performed. Any thread forked by other threads within the same frame is placed in the LCV rather than in the RCV. The set of threads executed in a single frame activation is called a *quantum*.

The TAM model enhances the parallelism of programs further by non-strict execution. Non-strict execution allows functions or arbitrary expressions to begin execution and possibly return results before all operands are computed. Non-strict execution also requires data structures able to be accessed while components are still being computed. In the TAM model, global data structures are based on I-structure [2] semantic, which provides synchronization on a per-element basis.

Several primitives are defined in the TAM model for fine-grain multithreading. They are operations for message transfer (SEND, RECEIVE), frame management (FALLOC, FFREE), scheduling (SWAP, STOP), thread generation (FORK, SFORK, SWITCH, POST, SPOST), and I-structure management (IALLOC, IFREE, IFETCH, ISTORE). Those primitives, combined with ALU operations, form an intermediate language called TL0.

3 Reducing Overheads of Local Communications

We have identified three cases where communication is involved between subtasks; message sending, parallel function invocation, and I-structure accesses. Each case will be discussed in detail in the following subsections.

3.1 Message Sending

In our model of fine-grain computation, messages are primarily used to carry arguments and results between function activations. The TAM model uses a mechanism called *active messages* [3] for fine-grain communication. In active messages, each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as the arguments. The role of the handler, or inlet, is to get the message out of the network and process the message by posting an appropriate thread.

For a message to be delivered, data should be copied from registers or frame slots to the network output buffer at the source node and from the network input buffer to frame slots at the destination node. Although the same mechanism can be used for local communications by providing a feedback path between the network input and output buffer, it is inefficient because the network interface becomes complex and unnecessary copying of data is performed.

To overcome these problems, we implement the LCV as a stack and extend it to be used for the linkage between sender and receiver in case of a local communication. For every inlet, we generate another version of codes, which extracts data from the LCV rather than from the network input buffer. We call this a *pseudo inlet.* Before sending a message, the program checks if the destination frame resides in the same node or not. If the communication is local, the program pushes arguments into the LCV with the address of the pseudo inlet. When the current thread reaches STOP, it fetches another enabled thread from the LCV, which is a pseudo inlet. The posted thread from the pseudo inlet is also put into the LCV. Therefore, the pseudo inlet and posted threads are executed within the context of the current frame without switching to the destination frame.

Frame accesses from threads or inlets are relative to the base address of the frame, or the frame pointer. In the original TAM model, all the threads in the LCV belong to the same frame. So the frame pointer is initialized only once when a frame switching occurs. On the other hand, the LCV in the proposed model holds the addresses of threads and pseudo inlets from different frames. Therefore, it is necessary to keep the frame pointer in the LCV and to initialize it whenever a thread is fetched from the LCV.

Table 1 shows an optimized implementation of SEND. **me** denotes my node number. **Node**() and **PseudoAddr**() are macros to find the node number for a given frame pointer, and an address of the pseudo inlet, respectively. At the beginning of a code block, there is a jump table that maps an inlet number to an actual address. Because a frame holds the base address of the corresponding code block, it is possible to find the address of an inlet or a pseudo inlet using a frame pointer and an inlet number.

Table 1. Optimized implementation of SEND

```
SEND dest_fp, dest_inlet, $arg_0$, ..., $arg_n$

    if (Node(dest_fp) == me) {
        Push $arg_0$, ..., $arg_n$ into the LCV;
        Push dest_fp, PseudoAddr(dest_fp, dest_inlet) into the LCV;
    } else {
        Store dest_fp, dest_inlet, $arg_0$, ..., $arg_n$ to the network output buffer;
        Send a message;
    }
```

3.2 Parallel Function Invocation

TAM's function invocation consists of two phases. In the first phase, the caller sends a request for frame allocation using a FALLOC operation. The callee allocates a frame upon receiving the request, initializes the frame, and then returns the frame pointer back to the caller. In the second phase, if the caller receives the callee's frame pointer, it sends arguments to predefined inlets. Due to the non-strict execution, arguments are sent one by one as soon as its value is known to the caller.

Actually, FALLOC is handled by a run-time system (RTS). If a user issues a FALLOC, the RTS in the source node determines the node where the function is assigned, and then sends a request message to the RTS of the destination node on behalf of the user. In the destination node, the RTS allocates a frame and schedules inlet 0 with the caller's frame pointer, the return inlet number and the new frame pointer as the arguments. Inlet 0 initializes the frame and returns its frame pointer to the caller.

Normal messages are used to return the new frame pointer or to send arguments and results. Therefore, they can be avoided using the optimization described in section 3.1 if the callee is allocated to the same node as the caller. However, the request message sent by an RTS also should be avoided in case of a local invocation. This can be done by revising an RTS routine of FALLOC, as described in Table 2. If the callee is local, the RTS adjusts the LCV so that the pseudo inlet 0 of the new frame can be executed after the current thread. In Table 2, `$fp` denotes the register which holds the current frame pointer.

Table 2. Optimized implementation of FALLOC

```
FALLOC code_block, return_inlet

Determine target_node, where the code block is assigned.
if (target_node == me) {
    Allocate a frame for code_block;
    new_fp ← base address of the new frame;
    Push $fp, return_inlet, new_fp into the LCV;
    Push new_fp, PseudoAddr(new_fp, 0) into the LCV;
} else
    Send a request message to the RTS of target_node;
```

3.3 I-structure Accesses

I-structures are accessed by split-phase operations such as IALLOC, IFREE, IFETCH and ISTORE. IALLOC and IFREE allocate and deallocate I-structures and IFETCH reads an element by sending a message to the node containing the data which returns the value to an inlet. In particular, reads of empty elements are deferred until the corresponding write occurs. ISTORE writes a value to an element, resuming any deferred readers.

Table 3 shows an optimized implementation of IFETCH, which removes any local message if the element resides in the same node and if it has valid data. **Tag**() indicates whether the word contains data (FULL), or not (EMPTY), or it has any deferred readers (DEFERRED). Actual data can be accessed using the macro **Data**(). IALLOC, IFREE, and ISTORE also can be optimized similarly.

Table 3. Optimized implementation of IFETCH

```
IFETCH return_inlet, heap_addr, element

if (Node(heap_addr) == me)
    switch (Tag(heap_addr[element])) {
        case EMPTY:
        case DEFERRED:
            Insert <$fp, return_inlet> to the deferred list;
            break;
        case FULL:
            Push Data(heap_addr[element]) into the LCV;
            Push $fp, PseudoAddr($fp, return_inlet) into the LCV;
            break;
    }
else
    Send a request message to the RTS of Node(heap_addr);
```

4 Experimental Evaluation

We have constructed an instruction-level simulator to evaluate the efficiency of the proposed optimization. The simulator was based on SPIM [4], an instruction-level simulator for MIPS instruction set, and was extended to parallel and multithreaded environments using a commercial event-driven simulator, SES/Workbench [5].

We have also implemented a translator which converts a TL0 program to MIPS assembly codes. The generated code consists of MIPS instruction set, assembly directives, and several system calls. System calls are used to transfer control to the RTS for some multithreading primitives. Actually, the simulator handles these primitives as if there were an RTS on every node. The translator also performs register allocation, because TL0 language uses memory-to-memory ALU operations. For experiments, the translator generates two versions of codes for a given TL0 program; one without optimization (ORG), and the other with optimization (OPT).

Table 4. Benchmark programs

Benchmarks	Arguments	TL0 lines	Memory Sizes (KB)		
			ORG	OPT	% Increased
`fib`	20	361	1.88	2.55	35.6
`qs`	500	2773	16.65	21.84	31.2
`mmt44`	100.0	3964	25.93	34.20	31.9
`dtw`	100.0	3500	24.62	34.92	41.8
`speech`	10240 30	6766	45.98	60.83	32.3
`paraffins`	17	10324	66.90	88.11	31.7

The interconnection network is not simulated in detail. Instead, we assume that the network has a uniform communication latency of 100 instruction cycles. The local feedback of a message is also assumed to take 10 cycles.

Table 4 shows arguments and program sizes of six benchmark programs used in the experiment. These applications are originally written in Id, and compiled to TL0 programs by the TAM group[1].

We can observe that the code size increases by 34.1% on average when the optimization is performed. Additional pseudo inlets primarily contribute to the increase in the code size. Secondary factors include codes to check if the destination is local and codes to push arguments and the address of a pseudo inlet into the LCV in case of a local communication.

Figure 1 gives a relative speedup when the optimization is used. Optimized codes always result in shorter execution times in spite of added cost of checking the destination of messages for every SEND, and saving and restoring the frame pointer for all primitives that access the LCV. Most of the benchmarks except `fib` and `paraffins` follow the general trend that the relative speedup decreases as the number of nodes increases. This is because the number of allocated frames per each node decreases, reducing the possibility of local communications. On average, optimized codes run 1.55 times faster than unoptimized codes on a single node, where all communications are local. There is 35.6% improvement in total execution times when we average six benchmark programs from 1 to 32 nodes.

It can be easily found that the benefits of our optimization mainly come from eliminating local messages. Figure 2 shows the ratio of the total number of messages generated from optimized codes with respect to that of unoptimized codes. Note that no message is generated with a single node

[1]They are freely available by anonymous ftp at `ftp://ftp.cs.berkeley.edu/ucb/TAM/idtam-0.3.tar.Z`

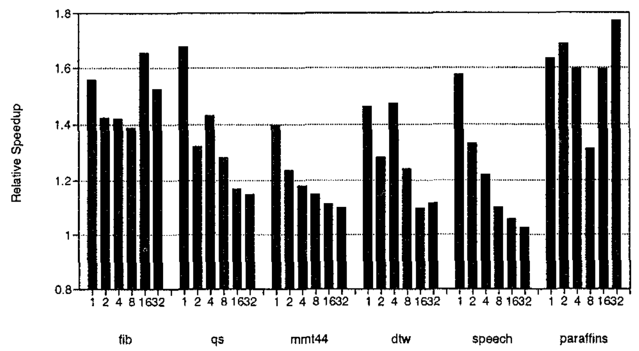

Figure 1. Relative speedup

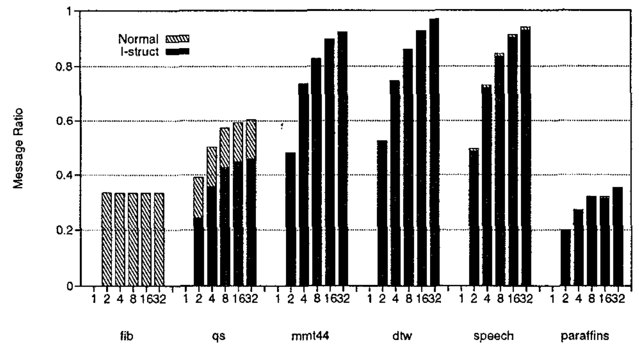

Figure 2. The reduction in total messages

using the optimized codes. The relatively low speedup of `mmt44`, `dtw`, and `speech` (see Figure 1), especially for the large number of nodes, is closely related to the low reduction in the number of messages.

We classify each message as *Normal* if it is generated by FALLOC or SEND, and as *I-struct* if it comes from I-structure operations. It is apparent from the figure that the number of I-structure messages is dominant. The reason that I-structure messages are not eliminated well in `mmt44`, `dtw`, and `speech` can be found from Table 5, which shows a dynamic statistics on I-structure operations.

Table 5. Statistics on I-structure operations

Benchmarks	IALLOC			IFETCH	ISTORE
	N	S	S/N	N	N
`fib`	0	0	-	0	0
`qs`	7885	15770	2.0	26540	14770
`mmt44`	10	30023	3002.3	507771	30023
`dtw`	14	40032	2859.4	2061000	40032
`speech`	157	45349	288.8	1565186	34909
`paraffins`	188272	615514	3.3	298632	608295

N denotes the total number of I-structure operations. S and S/N indicates the total size and the average size of I-structure elements requested by IALLOC, respectively. `mmt44`, `dtw`, and `speech` has a very large value of S/N, meaning that the large number of I-structure elements are allocated at once on a specific node. Because I-structures are concentrated on a few nodes, most of IFETCH requests are remote. Issues of the optimal and balanced distribution of global data structures are beyond the scope of this paper and should be addressed as a separate research topic.

5 Concluding Remarks

In this paper, we have presented an optimization technique to reduce overheads of local communications in fine-grain parallel computation. Although the optimization increases the code size slightly, the code size hardly affects the total memory requirement because frames demand much more memory at run time.

The advantage of suggested optimization can be summarized as follows. First, the network interface can be simpler because local feedback path need not be provided. In addition, the number of total messages which should be handled by the network interface is significantly reduced. Second, the cost of a local communication can be reduced by avoiding unnecessary copying of data to and from the network buffers. Third, the scheduling cost can be reduced by executing pseudo inlets and locally posted threads within the context of the current frame.

From the experimental results, we have observed that it is important to distribute I-structures in a balanced way for better speedup. This problem can be alleviated by employing an I-structure cache [6]. Note that the suggested optimization can also be used with such I-structure caches by eliminating the need for sending a message if the designated I-structure element can be found on a local cache. We expect greater speedup by combining the proposed approach and I-structure caches.

References

[1] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "TAM - A Compiler Controlled Threaded Abstract Machine," *J. of Parallel and Distributed Computing*, pp. 347–370, Jun. 1993.

[2] Arvind, R. S. Nikhil, and K. K. Pingali, "I-Structures: Data Structures for Parallel Computing," Tech. Rep. CSG Memo 269, MIT, Feb. 1987.

[3] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *Proc. 19th Int'l Symp. on Computer Architecture*, pp. 256–266, 1992.

[4] J. R. Larus, "SPIM S20: A MIPS R2000 Simulator," Tech. Rep. #966, University of Wisconsin-Madison, 1990.

[5] Scientific and Engineering Software Inc., *SES/Workbench 3.0 User's Manuals*. 1995.

[6] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of Cache Memories for Multi-Threaded Dataflow Architecture," in *Proc. 22th Int'l Symp. on Computer Architecture*, pp. 253–264, 1995.