Stream Processing Approach on the Fuce System for parallelizing Nested Loops with Data Dependency

Satoshi Amamiya, Makoto Amamiya
Department of Informatics, Faculty of Information Science and Electrical Engineering, Kyushu University

Abstract—It is widely known that nested loops with data dependency can be parallelized by transforming the original loops by techniques such as loop skewing, loop interchange (so called software pipelining and/or wavefront method). However, these loop transformation techniques need to solve the equation of simultaneous inequalities and modify loop index variables by ”human”. These procedures require complicated formalities to programmers. We have developed a much simpler method to parallelize nested loops with data dependency, which is an straightforward application of asynchronous handshaking stream programming on the continuation-based multithreading.

In this paper, we discuss our new approach in detail and show the enough performance enhancements can be exploited by our approach by evaluating the parallelized version of nested loop programs, on the Fuce runtime system on a commodity machine.

Keywords: stream processing, multithreading, loop parallelization

1. Introduction

The importance of high performance computing (HPC) have been widely recognized in the world and a lot of the researches on HPC have been conducted in past decades. High performance scientific computations are mostly large array operations. The speedup of array operations relies on basically the parallelization of multi-level nested loops for array. A variety of techniques for the loop parallelization was developed. In particular, for nested loops with data dependency, parallelization methods such as loop skewing, loop interchange (so called software pipelining and/or wavefront method) are noted. However, these loop transformation techniques need to solve the equation of simultaneous inequalities and modify loop index variables by ”human”. These procedures require complicated formalities to programmers.

We need a simpler approach to the parallelization and the programming techniques. We researched and developed the parallel architecture called Fuce[1] and its programming language CML (Continuation-based Multithreading Language) based on the idea of continuation and non-interruptible thread. The CML is designed for writing low-level event/demand-driven concurrent processes, stream processing, mutual exclusions, memory management and so on in a unified form.

In this paper, we introduce the stream processing based on the asynchronous handshake multithreading. And then we discuss the simpler parallelization technique for nested loops with data dependency as an application of our stream processing approach. Also we describe the implementation of the Fuce runtime system for commodity machines and show evaluation results of the stream processing programs executed on the Fuce runtime system.

2. Conventional Method for Parallelization

The wavefront method[2] is known as a typical method for parallelizing multi level nested loops with data dependency. We briefly explain the wavefront method using the following sample program[3]:

```
for (i = 2; i <= 6; i++)
  for (j = 2; j <= 6; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4
```

To eliminate data dependencies inside the loop, first we have to do loop-skewing and then interchange the outer and the inner loops. The transformed program looks like:

```
for (j = 4; i <= 12; j++)
  for (i = max(2,j-6); i <= min(6,j-2); j++)
    a[i][j-1] = (a[i-1][j-i] + a[i][j-1-i] + a[i+1][j-i] + a[i][j+1-i]) / 4
```

After a modification of this program with support of a parallel programming library, we execute the program in data parallel manner on parallel machines.

However, we have to solve a simultaneous inequality on variable i and j by hand to find the range of index variable i and j. We think this is not always a simple method. In addition, the users or programmers of such a wavefront program would not obtain high performance computation if they are not so familiar with a parallel programming library.

The idea of wavefront method itself seems to be simple and beautiful. But a implementation of a data parallel program based on this idea will become complicated. If we devise a different approach from the data parallelization, We will benefit from the basic idea of wavefront method without complicated procedures.
3. Stream Processing and the CML language

We introduce a special language called CML (Continuation-based Multithreading Language) to understand a basic behavior of stream processing. The large portion of the CML is derived from the ANSI C language. Some new grammatical constructs are designed and extended for stream processing. In this section, we give an outline of the CML is described in the papers[4], [5].

3.1 The Language CML

On the stream processing, temporal data storage space is usually set up between the sender and the receiver of data. Normally it is a bounded FIFO buffer. Here, for this purpose, we define a simple buffer storage called channel which can store basically only one data.

Channel Struct

```c
struct chan {int flag, value, from, to;}
```

The stream is closed if flag==0. A variable value holds a single stream data. The type of value is not only integer type, but also float or double is acceptable. We just re-define a new chan type as usage. A variable from holds sender process id and a variable to holds the receiver process id.

Process Definition

```c
process P(chan *ch, ...) <2> {...}
```

*ch is a pointer to a channel struct. <2> is the number of fan-in of process P. If this notation is abbreviated, the number of fan-in is 1. {...} is a body of process P.

Here, the number of fan-in of a process means the number of data which the process requests to send from other processes. Therefore, Once the process is scheduled and the value of fan-in of the process is initialized, the value of fan-in is simply decreasing in each data arrival. When the value of fan-in becomes zero, the process becomes ready to run 1.

Instance Creation

```c
int p = new P();
```

A data area and instance of process P are reserved.

Track-laying of Stream

Set the variable from in channel struct to the data sender process id and set the variable to to the receiver process id.

Process Invocation

When invoking a process, pass the channel name as actual parameter. And use mode indicator <+> for a sender process, <-> for a receiver process.

For example:

```c
p(ch1<+>, ch2<->));
q(ch1<-, ch2<+>);
```

Here, the process p sends a data to the process q via channel ch1 and the process q sends a data to the process p via channel ch2.

3.2 Basic Behavior of Stream

Consider a pair of processes (w, r) connected by a channel ch and that w generates a series of values for x, and sends them to r. This stream processing can be written in CML as in Figure 1. The action of w and r is controlled by cont instructions. Here we call init-, trigger-, or ack-cont for each cont instruction according to its usage. Init-cont is issued to invoke a newly created process instance. Trigger-cont is issued to invoke or prompt to invoke the other processes. Ack-cont is issued to the other process by the process which becomes ready to receive the next data.

When W is invoked by an init-cont or an ack-cont issued by R, W puts a value of x to ch, then issues a trigger-cont to R so that R can get the value. When R is triggered by W, R gets the value of x from ch, then issues an ack-cont to W so that W can put the next value. Note that the number of fan-in for W(R) is two since it requires a recur-cont plus a trigger-(ack-)cont.

It never happens that both W and R run at the same time, and that ch is accessed by both. Moreover, W and R run alternately, and it never happens that W puts values to ch in succession, and that R tries to get the next value from ch before another data is put.

This behaviour is easily understood by using a Petri net-like graph depicted in Figure 2. A token represented by a black disk corresponds to an issue of cont instruction. The token in the right arrow entering w is an init-token, the tokens in the left arrows of w and r are recur-conts.

3.3 Dynamic Growth of Stream

In the examples in section 3.2 the number of processes is small and fixed. They may be unworkable in the real world. In general, the number of processes will increase dynamically. The length of stream will grow accordingly.

Consider a process named Create which manages a process creation. Create process is written in CML for example as following:

```c
process create(chan *ch) <2> {
    darea chan ch1;
    new RW(ch<+>, ch1<+>);
    new create(ch1<->);
}
```

Here, a storage for a variable declared as darea is dynamically allocated at run time. After Create process creates
process main() {
    w = new W(); r = new R();
    ch->from = w; ch->to = r;
    w(ch<+>); // w's recur-cont
    r(ch<->); // r's recur-cont
}

process W(chan *ch) <2> {
    /* computing x's value */
    ch->value = x;
    cont ch->to; //trigger-cont to r
    recur;
}

process R(chan *ch) <2> {
    x = ch->value;
    cont ch->from; // ack-cont to w
    /* using x's value */
    recur;
}

Figure 1: A CML code for a basic stream processing.

Figure 2: Petri net for a simple stream processing among two processes.

and invokes a process RW, it re-creates itself 2. The second parameter of RW process is ch1<+> and the parameter of the new Create process is ch1<->. Therefore RW process and Create process form a stream processing among two processes as seen in Figure 1. The notation <*> added to the first parameter of RW process means that the channel is used for both of sending and receiving.

We define process RW in which the behavior of process W is added to process R in Figure 1. The CML code is as follows:

process RW(chan *in, chan *out)<3>{
    int x;
    in ?? x; // read from channel
    out !! x; // write to channel

    recur;
}

Here, 'in ?? x' and 'out !! x' are respectively the syntax sugar of
'x = in->value; cont in->from;','
'out->value = x; cont out->to;'.
The operators ??,!! are derived from CSP[6]. The number of fan-in of process RW is three because process RW is the sender as well as the receiver.

Next, in the definition of process main we replace the creation and invocation part of process R with them of process Create as follows:

process main() {
    int w = new W(ch<+>);
    new create(ch<->);
    cont w;
}

By this modification, process W and Create form a stream processing among two processes. And as mentioned above, process RW and Create also form a stream. When process Create is invoked, it creates process RW then pass a received data to process RW immediately. So data generated by process W are sent indirectly to process RW. Meanwhile process RW sends the data to process Create as soon as it receives. As the result as long as process W generates data, process RW is kept on being created and the size of the stream keeps on growing. Both ends of the stream is process W and Create. Between them there are myriad of process RW (refer to Figure 3).

4. Parallelization of Nested Loop

4.1 Parallelize Two-Level Nested Loop

We introduce local (instance) variable i to process W, j to process RW and Create which are defined in the section 3.3. When process W is invoked it increments the variable i and send the value of i (indirectly) to process RW via channel ch. Process Create increments variable j and creates process RW and Create with the value of j. Process RW keeps the value of j which is given at its creation.

Variables added to a parameter list of a process definition are treated as instance variables. For example, process Create is defined as follows:

process create(chan *ch, int j) <2>{
    darea chan ch1;
    j++;
    new RW(ch<+>, ch1<+>, j);
    new create(ch1<+>, j);
}

Figure 3 shows the behavior of three kinds of the processes W, Create, RW which are completely moving in sync. In this figure the longitudinal direction shows passage of time, the lateral direction shows growth of the stream. Note that all these processes on the stream are moving in parallel.
As mentioned above, process RW is kept on being created between process W and Create with time. The caption such like <1,2> in the figure is a pair of a data which is received from a immediate left process and the value of the instance variable j of the process RW. A data process RW receiving is initially sent by the process W, that is, the value of variable i. We label the process immediate right of process W as RW0. The initial value of j of process RW0 is set to 0. A data received from process W (that is, the value of i) is incremented with time. The value of j of process RW1 (RW0’s neighbor) is set to 1. A data RW1 received from RW0 is the data which the process W sent two clocks before. Therefore the value of the data RW1 received is just as small by 1 as that of RW0. In like wise manner, at the clock t, the pair of values process RWn holds is <t-n, n>.

Now we consider the pair of values as a element of two dimensional array and observe every pair of values process RW holds at every clock. Array elements referred at same clock on the stream is completely same as the elements referred by the wavefront method[2].

That is, by using a stream processing driven by only three kinds of processes W, RW and Create, it is possible to parallelize a two-level nested loop for two dimensional array. The following loop program in C is mostly equivalent to the above stream processing:

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    ... A[i][j]...; // array access
  }
```

4.2 Parallelize Nested Loop with Data Dependency

Consider the following nested loop:

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    A[i][j] += A[i+1][j-1];
```

In this case there is a data dependency among a loop. To eliminate these kind of data dependency we have to change the direction of the wavefront movement. Figure 4 shows a movement of wavefront on a two dimensional array. The dotted line is the original wavefront. A data dependency(red arrow) exists just on the original wavefront. If the direction of the wavefront can be changed as the solid line, a data dependency disappears. To obtain the safe direction of the wavefront, we slow the speed of i-direction down by half of the speed of j-direction. When we reduce the supply of data (the value of i) passed on the stream by half per clock, the speed of i-direction is reduced by a half.

So, we introduce a dummy process D immediate left of process W and make process W trigger not only process RW0 but also process D. Process D triggers process W immediately when D is invoked. The number of fan-in of D is 2 and that of W is changed to 3. These modification make the speed of data supply reduced by a half because process W can run every two clocks. Figure 5 depicts the safe stream processing. Running or stopped processes is shown as white circle or gray circle respectively. A straight or curved arrow indicates a trigger-cont or a ack-cont respectively. A recur-cont is abbreviated in this figure.

While a half of processes is stopped at the same time as shown in the figure, the maximum parallelism of the stream is limited to a half number of all processes.

By the way, as we mentioned in section 3.2, in fact neighboring two processes cannot run at the same time. Therefore when we write a stream processing code naturally in CML, we get the safe stream processing like Figure 5. This is the default behavior of our stream processing approach. Such kind of data dependency disappears automatically. We don't need to care about.

Next, consider a data dependency in a loop as following:

```c
A[i][j] += A[i+1][j-2];
```
The direction of the wavefront have to be changed also in this case. We introduce two dummy processes on left-side of process W. And each process issues ack-cont to not the neighbor process but after the next process. As a result, process W can only run every three clocks, the speed of data supply is reduced by one third. The direction of the wavefront changed to the way which satisfies $i : j = 1 : 3$.

4.3 Parallelize Three-level Nested Loop

With a small expansion of stream processing approach to a two-level nested loop, it is not so difficult to parallelize a three-level nested loop.

We modify the definition of process RW for a two-level nested loop so that process RW can act as same as process W and can make links to two different Create processes. Then two different streams will grow from a single process RW.

Figure 6 depicts a stream processing for a three-level nested Loop. A process on the crosswise stream named $RW_k$ and that on the lengthwise stream named $RW_j$. The process $RW_k$ sends the value of $i$ which is passed from the process W to lengthwise and crosswise. The behavior of process $RW_j$ is mostly as same as the process RW defined for a two-level nested loop. The wavefronts are generated as many as the number of the process $RW_j$ and move continuously on a two dimensional array. The following three-level loop is mostly equivalent to the stream processing explained in this section:

```c
for (i = 0; i < L; i++)
    for (k = 0; k < M; k++)
        for (j = 0; j < N; j++)
            ...A[i][k] = A[k][j]...
```

By the way, if a new stream grows from $RW_j$, this stream processing corresponds to a four-level nested loop. In like wise, if every process on every stream has the ability to generate a new stream, a n-level nested loop can be parallelized by our stream processing approach.

5. Fuce Runtime System

We implemented the Fuce processor as a software runtime on commodity OSes and parallel machines to judge the effectiveness of the idea of non-interruptible threads on Fuce.

5.1 ACM

The Fuce program is written as a set of functions. Each function is programmed using threads, and the corresponding function instances are executed in its runtime. Information of function instances is stored in a special storage called Activation Control Memory (ACM). Figure 7 depicts the structure of ACM. The information for controlling thread execution; sync-count, fan-in, code-entry and lock-bit, are stored in ACM.

5.2 TAC and TE

In this runtime, multiple thread execution engines are implemented. The key components of Fuce to support concurrent thread execution are the Thread Activation Controller (TAC), multiple Thread Execution units (TEs).

Each TE holds a ready thread queue independently. TE has the ability to steal a thread from the queue of another TE when its queue is empty. In order to realize this ability
efficiently, we use ABP Deque[7] which is one of simple lock-free algorithm and can be used as a queue or a stack. With ABP Deque, a ready thread is pushed to the deque as a stack without any mutual exclusion. A thread is popped by the TE from its deque. If the deque is empty, the TE try to steal (dequeue) a thread from another TE's deque. Figure 8 shows multiple TEs accessing simultaneously to the ACM.

The main part of TE is implemented in C as following:

```c
int thid;
Te_Env *env = (Te_Env *) GET_TE_ENV(thid);
while (!fuce_halt) {
    if (env->direct != NO_THREAD) {
        thid = env->direct;
        env->direct = NO_THREAD;
    } else {
        thid = pop(env->deque);
        if (thid == NO_THREAD)
            thid = steal_thread();
    }
    /*get thread info from ACM*/
    acm_page *page = &ACM[PAGE_NUM(thid)];
    thread_t *thinfo = &page->entries[OFFSET(thid)];
    void *da = page->darea;
    uint id = thinfo->id;
    Thread thr = thinfo->thread;
    thinfo->syncc = thinfo->fanin;
    /*invoke a thread as a func*/
    thr(id, da);
}
```

Each TE can be executed concurrently on commodity OSes when the above routine is setup as a pthread.

### 5.3 Handling Cont Signals

A cont instruction as in Figure 1 is handled by the runtime function `cont()` with a thread id as a parameter. When the size and address of a stack for a TE (as a pthread) are properly initialized according to the pthread specification, we can determine which TE calls the function `cont()`. For example, if stack size is set to 1MB and some local variable is defined, we can obtain the stack address by the following operation:

```c
#define GET_TE_ENV(ver) (((unsigned long) &ver) & (unsigned long) ~(0x100000 - 1L))
```

By using this macro the runtime function `cont()` can be simply defined as follows:

```c
void cont(thid id) {
    Te_Env *env;
    int syncc;
    thread_t *target = get_thread(id);
    ATOMIC_DECREMENT(target->syncc);
    if (target->syncc < 0)
        ERROR(...);
    if (target->syncc == 0) {
        /*get current TE’s env */
        env = (Te_Env *) GET_TE_ENV(id);
        if (env->direct == NO_THREAD)
            env->direct = id;
        else
            /* push thrd to ABP Deque*/
            push(env->deque, id);
    }
}
```

ACM is modified inside the function `cont()`. This modification must be done by a atomic operation because multiple threads executed in TEs would access the same location in the ACM at the same time.

### 6. Evaluation of Stream Processing

We evaluated the performances of stream processing CML codes on the Fuce runtime system. The program codes are transformed from two-level nested loop with data dependency. We used two benchmark programs, ISC’07 loop and the levenstein distance.

#### 6.1 ISC’07 Loop

This sample program was created when we discovered and published[5] the basic idea of stream processing on Fuce for the first time. The program code is as follows:

```c
for (i=1;i<N;i++)
    A[i][0] += A[i-1][0];
for (j=1;j<M;j++)
    for (i=0;i<N;i++) {
        if (i==0) A[i][j] += A[i+1][j-1];
        else if (i==N-1) A[i][j] += A[i-1][j];
    }
```

Pay attention to the second nested loop. A data dependency at the last else clause seems mostly as same type as that of example in section 4.2. We don’t need to care about safety. The data dependency disappears naturally when it is transformed to a stream processing. The first loop can be fused into the inner loop of the second nested loop.

#### 6.2 Levenstein Distance

The second benchmark program is the Levenstein distance algorithm, one of dynamic programming (DP) example.
Many other DP algorithms have similar structure to this program, the Levenstein code looks as follows:

```c
int A[M+1][N+1];
for (i=0; i<M; i++) A[i][0] = i;
for (j=0; j<N; j++) A[0][j] = j;
for (i=1; i<=M; i++) {
  for (j=1; j<=N; j++) {
    int cost =
      (str[i-1] == str[j-1]) ? 0 : 1;
    A[i][j] = minimum(A[i-1][j] + 1,
                      A[i][j-1] + 1,
                      A[i-1][j-1] + cost);
  }
}
```

The first two loops can be fused into the inner loop of the last nested loop. The data dependency of this program also disappears as well as that of ISC’07.

### 6.3 Performance Evaluation

The two transformed benchmark programs are executed on a Linux machine in which two Intel Xeon 3.3GHz processors is equipped. Each processor has six cores, in total twelve processor cores are utilized. Both transformed benchmark programs are configured so that input array size=16Kx16K, tile size=256 (256 array elements). Note that each process is defined with tiling[8] to operate 256 elements of array in a single process.

Figure 9 shows the scalability against the number of CPU cores for each program. On the basis of sequential execution times of the original programs, the speedups of the stream processing is shown on the graph. The performance of the levenstein is good enough. However that of ISC’07 reaches a ceiling around 8 CPU cores. We think that it is not so easy for the stream processing, especially for ISC’07 to utilize CPU caches sufficiently. Therefore the bandwidth of memory bus would be consumed away and it would interfere with communications between processes.

### 7. Conclusion

In this paper, we introduced the stream processing, which is based on the asynchronous handshake multithreading technique and explained in detail a new approach to the wavefront method.

With our approach, it’s possible to parallelize easily multi-level nested loops which hold strong data dependencies. Nested loops with data dependency appear frequently in dynamic programming algorithms. Many important time-consuming algorithms especially in biology fields are built up based on dynamic programming. We expect that the applications of our stream processing approach help to reduce time consumption of bioinformatic analyses greatly.

### Acknowledgment

This research was supported by the Ministry of Education, Culture, Sports, Science and Technology(MEXT), Grant-in-Aid for Scientific Research (B), 21300011

### References


