1 차 회귀 연산의 병렬처리

윤 영 하(1999059)　　**나 상 옥(1998509)**

한국정보통신대학원대학교

toreit@icu.ac.kr　**nso@icu.ac.kr**

Abstract

본 연구에서는 과학 연산 부분에서 많이 사용되고 있지만, 그 내부에 병렬성을 가로 막는 LCD(Loop Carried Dependence)로 인하여 일반적인 방식으로는 고속의 병렬처리가 불가능한 회귀연산에 대해 SMP 기반에서 병렬처리를 가능케 하는 방법을 소개하고 아울러 그 성능에 대해서도 살펴볼 것이며 아울러 MPI를 이용한 환경에서도 그 결과를 살펴볼 것이다.

Introduction

회귀 연산은 과학 연산부문에서 많이 사용되고 있는 계산의 형태이다. 이러한 계산 형태는 대부분 loop 를 이용해서 계산이 되고 있는데, 계산 자체의 특성상 LCD(Loop Carried Dependence)가 존재하게 되어 기존에 이용되고 있는 순차적인 수행코드의 병렬화를 가로막고 있는 주요원인 중 하나이다. 지금까지 많은 방식이 이 회귀연산의 병렬화를 위해 연구되어 왔지만 그 대부분은 Vector 컴퓨터에서의 이용을 전제로 하여 개발되어 SMP 기반의 컴퓨터에서는 Vector 방식을 적용하여서는 성능향상을 노릴 수 없었다. 하지만 현재 개별 CPU 의 성능은 비약적으로 발전하고 있고, 역시 SMP 기반의 컴퓨터는 그러한 발전에 힘입어 고속의 BUS 와 cache 의 적절한 이용으로 그 하드웨어 적인 성능이 향상되고 있다. 그러나 SMP 의 장점은 무엇보다도 공유메모리를 사용함으로 인한 프로그래밍의 용이함에 있다. 이러한 SMP 기반 위에서의 작업은 현재의 순차적 프로그램을 병렬화시키는데 있어서 MPP 나 다른 어떤 머쉰 타입보다도 유리한 면이 많다.

본 연구에서는 무엇보다도 1 차 회귀 연산에 대해 SMP 기반에서 thread 를 통한 병렬화 과정을 제시하고 이를 통해 회귀연산의 SMP 기반에서의 병렬화 가능성에 대해 살펴보고 아울러 그 성능에 대해서도 기존의 알려진 loop 처리와 캐쉬 이용률 향상 등의 방법을 이용하여 논의할 것이다. 또한 MPP 기반의 머쉰에서도 MPI 를 이용하여 그 성능을 살펴볼 것이다.

Preliminaries

$$x_n = \sum_{k=1}^{m} (a_{n-k} x_{n-k}) + c_n \quad (n > m)$$

이러한 형태의 연산을 회귀연산이라고 부르며 특히 *m=1* 일 때를 1 차 회귀 연산(first order linear recurrence)이라고 부른다. 이때 a_k는 상수(constant)일 수도 있다. 1 차 회귀 연산은 많은 경우의 linear equation, 행렬연산등에서 나타나고 있으며, 2 차 회귀 연산(second order linear recurrence)의 대표적인 예는 $X_n = X_{n-1} + X_{n-2}$의 피보나시 수열이다. 이와 같은 회귀연산의 가장 일반적인 computation 은 **그림 1** 과 같다.

```
Do i=1, N
        X(i)= A(i)*X(i-1) + C(i)
enddo
```

그림 1. 1 차 회귀 연산의 형태

이런 경우 X에 대해 LCD가 존재하게 되어 data decomposition이나 loop distribution을 통한 병렬화는 불가능해진다. 따라서 이러한 회귀 연산의 경우는 지금 까지 Vector 컴퓨터의 독특한 연산방식을 이용하여 그 성능을 향상시키거나, loop unrolling이나 pipeline 등의 적절한 이용[3]으로 성능을 향상시키는데 주된 연구가 이루어져 왔다. Vector 머쉰 상에서는 Regular-Schedule[2]과 같은 방법을 이용하여 효과적인 연산을 수행하고 있으며, 이를 처리하는 컴파일 기법역시 vector 기반에서[1]연구되고 있다.

그러나 SMP 환경에서 이러한 연산의 병렬처리에 대한 연구는 활발히 이루어지지 않고 있는 현실이다. 회귀 연산 자체가 과학 기술 등 한정된 분야와 대량의 자료를 처리하는 분야에서 이용되고 있기 때문에, 일반적인 2-way, 혹은 4-way 환경에서 이러한 대량의 자료를 처리하는 일은 드물었다. 그러나 MPP나, vector 머쉰에 이어 나온 CC-NUMA 머쉰은 SMP의 장점을 포용하면서 MPP 머쉰에 버금가는 연산능력을 지닐 수 있기 때문에 대용량의 자료처리와 과학기술분야에서 UMA(Uniform Memory Access)환경을 이용하여 병렬처리를 손쉽게 할 수 있는 환경을 제공하고 있다. 또한 회귀연산의 경우 현재의 발달된 컴파일러기술로 dependence의 검출과 이를 통한 cycle detection이 가능해짐에 따라 loop 안에서 형성되는 cycle을 회귀연산의 형태로 인식하는 일 또한 가능하게 되었다. 따라서 고속의 처리에 있어서 일반적인 경우 4-way, 혹은 8-way의 시스템을 구축하여 손쉽게 병렬처리를 통한 자료처리가 가능해지고 있다.

1차 회귀 연산은 일반적인 자동 병렬화 컴파일러에서는 LCD만을 표시하고 있기 때문에 순차적인 코드를 병렬 코드화 시키는 자동 병렬화 컴파일러[4]에서는 이 부분을 병렬처리 불가로 인식하여 순차적으로 실행시키고 있다. 따라서 이 부분에 대해 병렬처리를 하기위해서는 검출된 dependence와 이를 이용한 cycle의 검출 그리고 distance vector [5]인식 등의 선행작업이 필요하다.

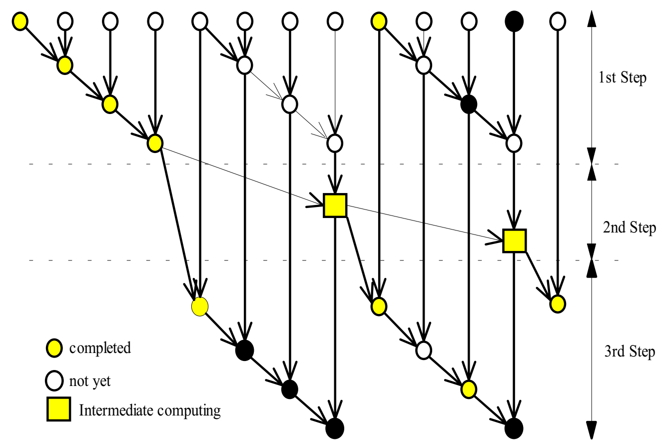


그림 2. 연산과정

Algorithm

회귀연산의 병렬수행에 있어 많은 연구가 되어 있지만, 그 대부분은 vector 기반에서 연구되고 구현된 것이라 알고리즘 자체가 SMP에 적용이 힘든 경우도 있으며, 또한 SMP 기반에서 thread를 생성시켜서 구현을 할 경우 그 성능향상이 이루어지지 않는 경우도 있다.

이 논문에서는 벡터처리의 고속화[6]에 소개된 방식 중 한 가지를 이용하여 알고리즘을 구현하였다.

알고리즘의 기본적인 동작 구조는 **그림 2**와 같다. 이 알고리즘은 모두 3단계의 작업을 통해 이루어진다. 첫 번째 작업에서는 각각의 coefficient에 대한 연산과 회귀 연산을 병렬로 partitioning하여 thread를 생성하여 수행하며, 두 번째 작업에서는 첫 번째 작업에서의 결과를 토대로 필요한 인자를 연산하여 각각의 thread로 보내주는데 이 부분은 순차적으로 실행된다. 그 후 세 번째 작업에서는 전달된 인자를 이용하여 다시 한번 연산을 하여 모든 X에 최종적인 결과가 들어가게 된다. **그림 3**은 두 번째 작업에서 중간결과를 구하는 식이다.

CPU의 개수가 m이고 배열의 크기가 n일 때, thread를 CPU에 바운드시켜 각각의 thread에 n/m개의 연산량을 할당시킨다.

$$
\begin{aligned}
& x_1 \\
& x_2 = a_2 x_1 + c_2 \\
& \ldots\ldots \\
& x_4 = a_{4,2} x_1 + a_{4,3} c_2 + a_4 c_3 + c_4 \\
& x_5 \longleftarrow a_5 + c_5 \\
& \ldots\ldots\ldots\ldots \\
& x_8 = a_{8,5} + a_{8,7} c_6 + a_8 c_7 + a_8 \\
& x_9 \longleftarrow a_9 + c_9 \\
& \ldots\ldots\ldots\ldots \\
& x_{12} = a_{12,9} + a_{12,11} c_{10} + a_{12} c_{11} + c_{12}
\end{aligned}
$$

$$
\begin{aligned}
& x_8 = a_{8,5}(x_8 - x_7)x_4 + x_8 - a_{8,5} \\
& x_{12} = a_{12,9}(x_{12} - x_{11})x_8 + x_{12} - a_{12,9}
\end{aligned}
$$

그림 3. Step 2 에서의 중간 값 구하기

그림 4~6 은 3 단계로 나누어진 작업의 흐름도이다. 첫

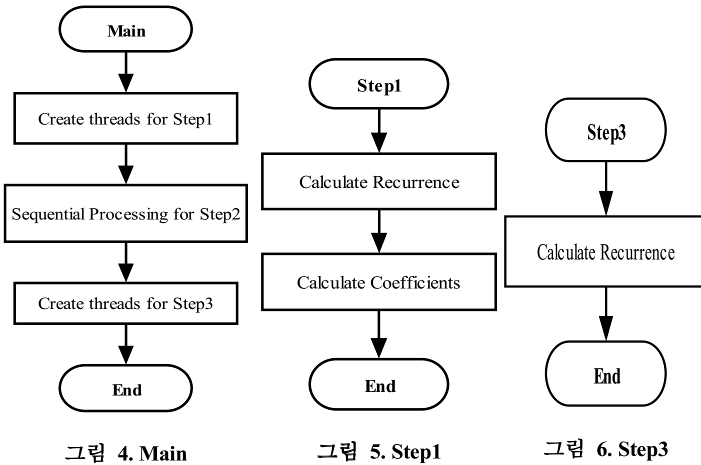


그림 4. Main **그림 5. Step1** **그림 6. Step3**

번째 작업에서는 1 차 회귀 연산에 대해 나누어진 작업에 대해 처리를 하는데, 2 번째 작업에서 각각의 coefficient 에 대한 연산이 필요하게 되므로, 실제적인 작업량은 multiplication 을 기준으로 하여 ***O(2n/m)***이 된다. 그리고 두 번째 작업에서는 그림에서와 같은 연산을 수행하여 세 번째 작업으로 건네질 값을 구하는데 ***O(2m)***의 작업이 필요하게 되고, 3 번째 작업에서는 회귀연산에 대해서만 연산을 취하므로 ***O(n/m)***이 된다. 따라서 총 연산량은 ***O(3n/m + 2m)***이 된다. 이 경우 수식상으로는 적어도 ***m ≥ 4*** 가 되어야만 성능향상이 이루어 질 수 있을 것으로 여겨지나, 실제로는 각각의 CPU 에 장착된 L2 캐쉬의 증가에 힘입어 그 연산성능을 고속화될 것으로 예상되었다. 또한 loop unrolling 을 적절히 사용하고, C 에서 제공하는 register 변수나, 임시변수를 사용하여 캐쉬적중율을 높임으로서 상당한 성능향상이 이루어질 것이다.

구현

C 와 thread 를 이용하여 구현을 하였는데, **Main** 에서는 thread 를 생성하여 **Step1** 에서 병렬 연산을 수행하고 결과를 순차적으로 연산해 다시 **Step3** 에서 병렬처리를 한다. **Step1** 에서는 **Step2** 을 위하여 각각의 coefficient 의 곱을 연산한다. 이를 **Step2** 에서 순차적으로 각각의 thread 에게 건네질 값을 연산한 후 다시 thread 를 생성하여 연산을 수행하면 최종적인 값이 나오게 된다.

Test 환경

Test 를 시행한 기기 SMP machine 2 가지와 MPI 를 이용한 MPP 방식의 기기 1 가지이다.

1) Pentium III Xeon 500 MHz 512K 4 CPU
   Solaris 2.6
   GNU C 컴파일러
   Solaris Thread library
2) SPARC 60MHz 4 CPU
   Solaris 2.6
   GNU C 컴파일러
   Solaris Thread library
3) CRAYT3E 400MHz 4,8,12,16 CPU
   MPI library

SMP 머쉰에서는 생성되는 thread 를 각각의 CPU 에 바운드시키기 위해 ***thr_setconcurrency()***를 이용하였으며, testing 환경은 5 가지로 나누어 진다. 회귀연산은 floating point multiplication 이고, *A, X, C* 배열 각각의 크기는 두개의 머쉰에서 동일하게 10,000,000 개로 구성되어

진다. MPP 머쉰에서는 4 개, 8 개, 12 개 16 개의 CPU 상에서의 동작을 비교했으며 자료의 크기는 640000 개의 double 형 이다. 비교에 사용되는 순차적인 연산은 조작을 가하지 않은 **그림 1** 과 같은 형태를 취한다.

1) 알고리즘만을 구현하여 loop 및 cache 에 관계되는 어떠한 처리도 행하지 않으며 compiler 옵션 역시 사용치 않는 경우
2) 1 개의 register 변수를 이용하여 write 되는 경우 이를 경유하게 하여 cache 효율을 높이며 컴파일러 옵션을 사용치 않는 경우
3) 2 개의 register 변수를 이용하여 write 되는 경우와 coefficient 의 연산에 이용하며 컴파일러 옵션을 사용치 않는 경우
4) 12 개의 loop unrolling 을 시행하고 cache 효율을 높이기 위해 write 되는 경우 temporal variable 을 이용하고 이를 register 변수로 처리하며 coefficient 의 연산도 register 변수를 이용하며 컴파일러 옵션을 사용치 않는 경우.
5) 12 개에 대해 loop unrolling 을 시행하고 cache 효율을 높이기 위해 write 되는 경우 temporal variable 를 사용하며 컴파일러 옵션을 사용치 않은 경우
6) MPP 머쉰에서 4 개,8 개,12 개,16 개의 CPU 를 이용하여 12 개의 loop unrolling 과 임시변수를 이용하는 경우와, loop 및 cache 에 대한 조치를 취하지 않은 경우

결과

1) 60MHz Sparc 4 Way

Non handled 로 되어 있는 그림은 1 번 환경, 이후 표에서

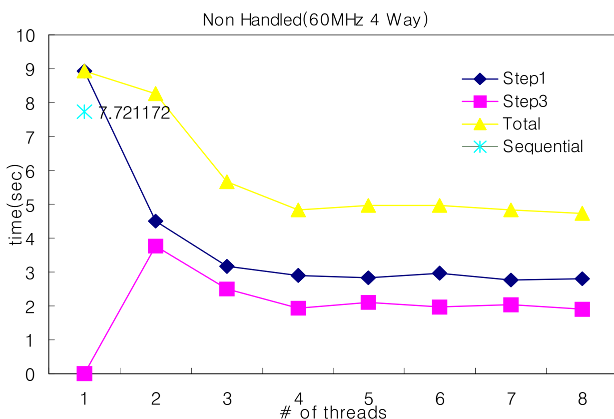
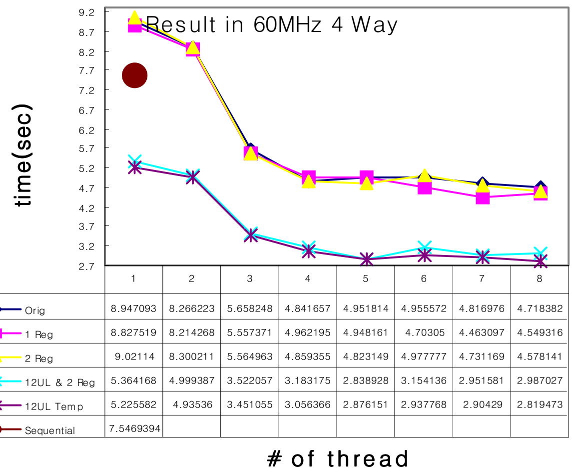


그림 7. Non handled Test



	1	2	3	4	5	6	7	8
Orig	8.947093	8.266223	5.658248	4.841657	4.951814	4.955572	4.816976	4.718382
1 Reg	8.827519	8.214268	5.557371	4.962195	4.948161	4.70305	4.463097	4.549316
2 Reg	9.02114	8.300211	5.564963	4.859355	4.823149	4.977777	4.731169	4.578141
12UL & 2 Reg	5.364168	4.999387	3.522057	3.183175	2.838928	3.154136	2.951581	2.987027
12UL Temp	5.225582	4.93536	3.451055	3.056366	2.876151	2.937768	2.90429	2.819473
Sequential	7.5469394							

그림 8. 60MHz 4-Way 에서의 결과

는 Orig 로 되어 있다. *1 Reg* 는 2 번 환경, *2 Reg* 는 3 번 환경, *12UL&2Reg* 는 4 번 환경, *12UL&Temp* 는 5 번 환경이다. Sequential 은 비교에 사용된 순차적 처리의 예이다.

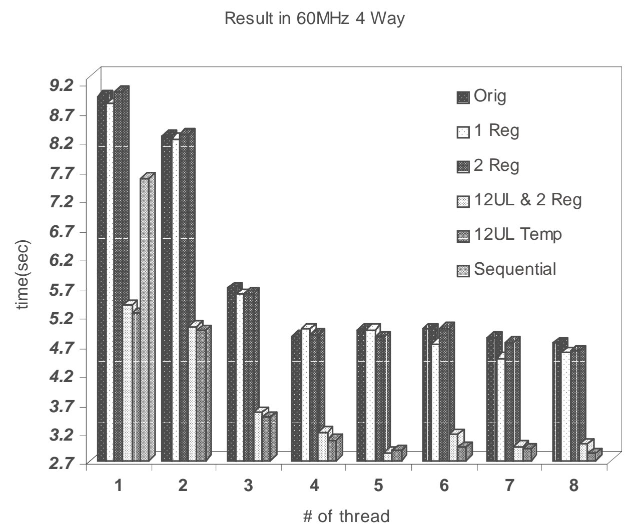


그림 9.각각의 성능비교

2) 500MHz Xeon 4 Way

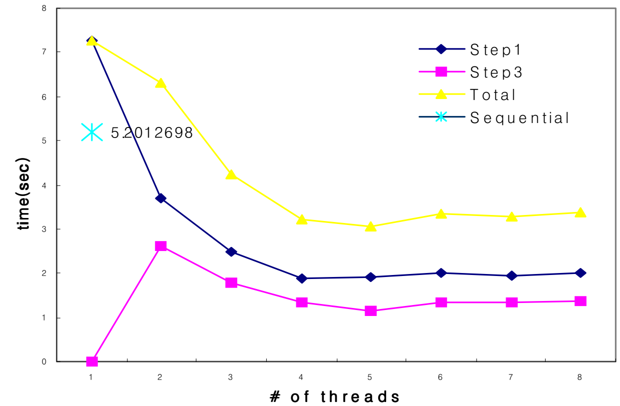
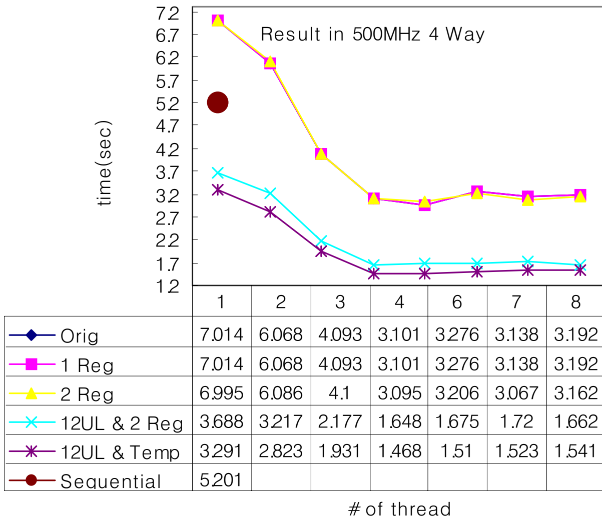


그림 10. Non handled Test



	1	2	3	4	6	7	8
Orig	7.014	6.068	4.093	3.101	3.276	3.138	3.192
1 Reg	7.014	6.068	4.093	3.101	3.276	3.138	3.192
2 Reg	6.995	6.086	4.1	3.095	3.206	3.067	3.162
12UL & 2 Reg	3.688	3.217	2.177	1.648	1.675	1.72	1.662
12UL & Temp	3.291	2.823	1.931	1.468	1.51	1.523	1.541
Sequential	5.201						

그림 11. 500MHz 4 Way 에서의 결과

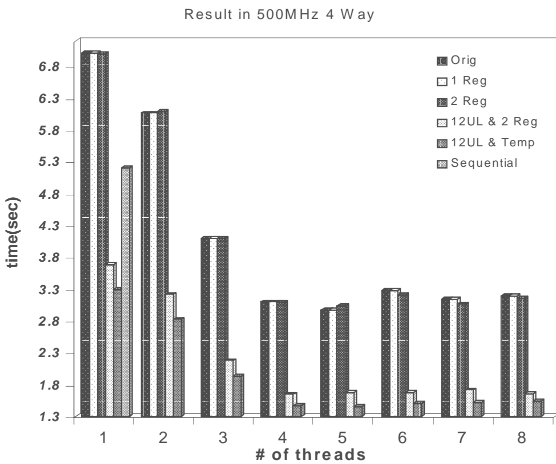


그림 12. 각각의 성능비교

4) CRAY T3E

CRAY T3E 에서는 메모리의 문제로 640000 개의 double 형 배열을 이용하여 test 를 했다. Orig 로 나온 부분은 알고리즘만을 구현한 부분이고, LU&Temp 는 12 번의 loop unrolling 과 임시변수를 사용한 부분이다. 결과는 성능향상이 없었다. SMP 와는 달리 MPI 를 이용하여 communication 하는 시간이 연산에 걸리는 시간보다 많은 것으로 보여지며 4 개의 CPU 뿐 아니라 8 개, 16 개의 CPU 를 이용했을 때도 순차적인 실행에서의 성능을 앞서지는 못했다. 또한 8 개에서 16 개로 나아갈 때 saturation 이 일어남을 볼 수 있다.

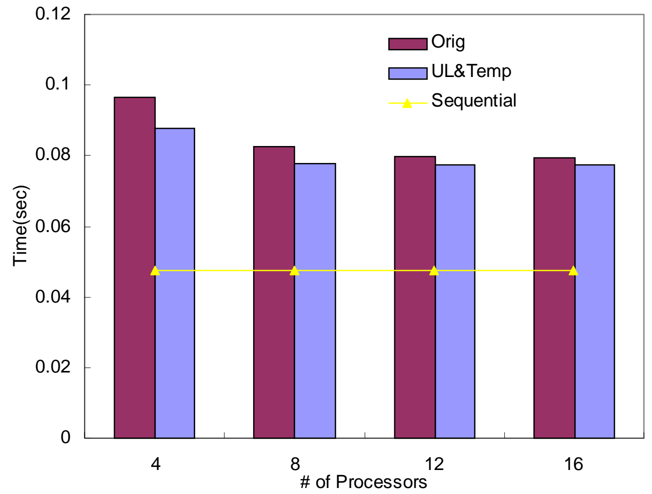
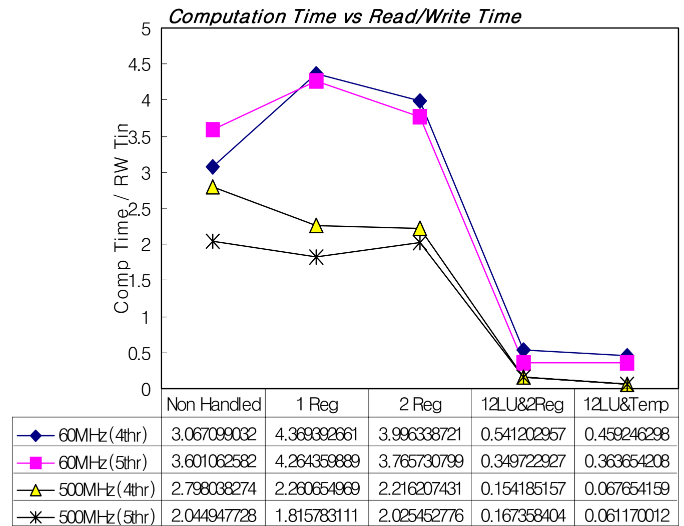


그림 13. CRAY T3E 에서의 결과

분석



	Non Handled	1 Reg	2 Reg	12LU&2Reg	12LU&Temp
60MHz(4thr)	3.067099032	4.369392661	3.996338721	0.541202957	0.459246298
60MHz(5thr)	3.601062582	4.264359889	3.765730799	0.349722927	0.363654208
500MHz(4thr)	2.798038274	2.260654969	2.216207431	0.154185157	0.067654159
500MHz(5thr)	2.044947728	1.815783111	2.025452776	0.167358404	0.061170012

그림 14. Computation Time 과 Read/Write Time 비교

MPI 를 이용하였을 때에는 computation time 이 communication time 을 감쇄시키지 못했음이 분명하므로 주로 SMP 의 경우에서 분석을 하겠다.

알고리즘의 Time Complexity 는 $O(3n/m + m)$이다. 따라서 메모리에 의한 효과 등 다른 영향을 제외하고 생각해 볼 때 이 알고리즘은 적어도 4 개의 CPU 가 있어야만 순차적인 실행 때보다 나은 결과를 보일 것이고 그 때의 Speed Up Factor[7]는 1.33 일 것이다. 그러나 실제의 수행에 있어서는 loop 처리를 하지 않았을 때도 Speed Up Factor 는 이론상의 수치를 넘어선다.

SpeedUp Factor					
	Non Handled	1 Reg	2 Reg	12LU & 2Reg	12LU & Temp
60MHz (4Thr)	1.594738	1.522349	1.538635	2.349745	2.454847
60MHz (5Thr)	1.559261	1.526667	1.550185	2.634674	2.608663
500MHz (4Thr)	1.617199	1.676006	1.68198	3.156626	3.550083
500MHz (5Thr)	1.707086	1.74694	1.710175	3.108655	3.586518

위의 결과로 볼 때 회귀연산에 있어서 연산수행 속도 못지 않게 메모리 접근 속도 역시 매우 중요한 요소중의 하나이다 Computation time : Read/Write time 을 비교해 보면 **그림 14** 와 같은데, 이때 초기 3 가지 방법에서는 상대적으로 Computation time 이 전체 시간에 있어 많은 양을 차지하고 있으나, loop 조작을 할수록 이 회귀연산은 연산보다는 메모리접근에 상대적으로 더 많은 시간을 소모하고 있음을 알 수 있다.

위의 결과에서 일부 5 개의 thread 를 생성하였을 때 근소하게 결과가 좋아지는 것은 알고리즘의 구성이 초기 Step 1 에서는 4 개의 thread 를 연산을 위해 생성하지만, Step 3 에서는 그보다 1 개 적은 3 개의 Thread 를 연산을 위해 생성하기 때문에, 5 개의 thread 를 생성하였을 때 Step 3 에서 모든 CPU 가 이용되므로 이런 결과가 나오는 것이다. 현재 가장 효율적으로 나온 500MHz 4 개의 thread 에서 12 번의 loop unrolling 을 하고 temporal 변수를 이용하였을 때의 speed up factor 가 3.5 로 상당히 좋은 편이지만, 비교대상을 loop unrolling 을 행한 순차적 실행으로 바꾸어 본다면 그 결과는 아주 좋은 편은 아닐 것이다. 하지만, **그림 11** 에서도 나타나듯이 CPU 가 증가할 때 그 성능의 향상이 만족할 만큼 일어나지 않는 것은 메모리와 연결 버스 상의 병목현상으로 생각된다. 그러나 이러한 맹점은 고속의 RAM 과 연결 버스가 개발되고 있으므로 앞으로는 이러한 약점은 해소되리라고 생각한다.

결론

SMP 는 예전에는 기계 자체의 구현과 비용상의 문제로 개발이 활성화되지 못했지만, 현재는 CPU 의 성능이 급속도로 발전하고 컴퓨터 구조에 대한 연구의 진척으로 성능면에서 많은 발전을 이루었다. 특히 SMP 만이 가지는 UMA 환경은 프로그램개발자에게 많은 편리한 환경을 제공함은 주지의 사실이다. 현재는 이러한 SMP 머쉰을 기반으로 한 CC-NUMA 머쉰이 등장하고 있고, 이러한 개발에 측면에서 MPI, PVM 을 이용한 MPP 기반의 연구, Vector 머쉰 상에서의 연구에 못지 않게 SMP 기반에서 CC-NUMA 를 이용할 수 있는 연구가 필요한 시기라고 본다. 본 연구에서 다룬 내용은 이러한 SMP 기반의 환경에서도 충분한 성능향상이 가능하며 또한 과학기술 등의 연산에 있어서도 충분히 적용을 시킬 수 있다는 가능성을 말해주고 있다. 이 연구에서 구현한 알고리즘과 그 성능은 아직 상당히 많은 개선점을 가지고 있다. 특히 thread 의 생성에 있어 Step 3 에서의 CPU 1 개가 남아 있는 현상이라던가, Step 1 과 Step 3 를 실행하면서 각각 별도의 thread 를 생성하는 것은 아무리 프로세서의 생성보다는 thread 의 생성 부하가 적다고 하여도 overhead 로 작용하는 것은 틀림없는 사실이다.

1 차 회귀 연산 뿐 아닌 기존의 MPP 나 Vector 머쉰에서 적용된 많은 알고리즘들이 SMP 기반에서도 충분히 적용될 수 있는 가능성이 있으므로 이에 대한 연구가 필요할 것이다.

Reference

[1] Y.Tanaka, K.Iwasawa S.Gotoo, Y.Umetani, *Compiling Techniques for First-Order Linear Recurrences on a Vector Computer*, IEEE 1988

[2] H.Wang, A.Noolau, S.Keungt, K,Siut, *Scalable Techniques for Computing Band Linear Recurrences on Massively Parallel and Vector Supercomputers*, IEEE 1994

[3] S.Carr, C.Ding, P.Sweany, *Improving Software Pipelining With Unroll-and-Jam*, IEEE 1996

[4] 한동수, *병렬Fortran 컴파일러 개발에 관한 연구*, 한국전자통신연구원, 1999

[5] M.Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996

[6] M.Nakamura, Y.Okabe, T.Tsuda, New Fast Algorithms for First-Order Linear Recurrences on Vector Computers, Fifth Workshop on Compilers for Parallel Computers(Malaga, Spain), pp. 167-174, July 1995

[7] B.Wilkinson, M.Allen, *Parallel Programmnig*, New Jersey, Prentice Hall, 1999