Memory-to-Memory 방식 벡터컴퓨터에서의 외연적 유한요소법의 벡터화

Vectorization of an Explicit Finite Element Method on Memory-to-Memory Type Vector Computer

이 지 호*
Lee, Jee Ho
이 재 석**
Lee, Jae Seok
김 문 현***
Kim, Moon Hyun

요 지

외연적 유한요소법은 벡터처리에 적합한 구조를 가지고 있어 벡터컴퓨터를 이용하면 기존의 스칼라 컴퓨터에서보다 훨씬 빠르게 해석을 수행할 수 있다. 본 논문에서는 memory-to-memory 방식의 벡터컴퓨터에서의 외연적 유한요소법의 효율적인 벡터화 방법을 제시하였다. 먼저 벡터컴퓨터의 구조적 특성과 무관하게 적용될 수 있는 일반적인 벡터화 기법을 고찰한 후 memory-to-memory 방식의 벡터컴퓨터에 적합한 벡터화 기법을 개발하였다. 개발된 벡터화 기법의 유용성을 확인하기 위해 외연적 유한요소 프로그램인 DYNA3D를 memory-to-memory 방식의 벡터컴퓨터인 HDS AS/XL V50에 이식한 결과 스칼라에 비해 2.4배 이상의 성능 향상을 얻을 수 있었다.

ABSTRSACT

An explicit finite element method can be executed more rapidly and effectively on vector computer than on the scalar computer because it has suitable structures for vector processing. In this paper, an efficient vectorization method of the explicit finite element program on the meomory-to-memory type vector computer is proposed. First, the general vectorization method which can be applied regardless of the vector architecture is investigated, then the method which is suitable for the memory-to-memory type vector computer is proposed. To illustrate the usefulness of the proposed vectorization method, DYNA3D, the existing explicit finite element program, is migrated on HDS AS/XL V50 which is the memory-to-memory type vector computer. Performance results on actual test show a vector/scalar speedup is above 2.4.

* 한국과학기술연구원 시스템공학연구소 연구원
** 한국과학기술연구원 시스템공학연구소선임연구원,정회원
*** 한국과학기술연구원 시스템공학연구소 책임연구원,정회원

이 논문에 대한 토론은 1991년 6월 30일까지 본 학회에 보내주시면 1991년 12월호에 그 결과를 게재하겠습니다.

1. 서론

벡터처리기(vector processor)는 거의 동시에 다량의 데이타를 처리함으로써 스칼라처리기(scalar processor)에 비해 월등히 향상된 계산능력을 제공하며 이런 능력을 이용하여 시간을 많이 소모하는 복잡한 공학 문제들을 해결하려는 노력이 여러 분야에서 진행되고 있다[1].

충돌해석과 같은 동적 비선형 문제의 수치해석은 계산량이 많이 요구되는 분야로서 컴퓨터의 계산능력이 큰 제약조건으로 작용해 왔다[2]. 외연적 유한요소법(explicit finite element method)은 내연적 유한요소법(implicit finite element method) 보다 식이 간단하여 빠른 수행속도를 보장하며 요구되는 기억용량도 적어 동적 비선형 문제의 해석에 적합하다[3]. 더우기 외연적 유한요소법은 벡터처리에 적합한 구조를 가지고 있어 벡터처리기를 이용하면 기존의 스칼라컴퓨터에서 보다 훨씬 빠르게 해석을 수행할 수 있다.

기존의 유한요소해석 프로그램에서 벡터처리기능을 이용하기 위해서는 특별한 변환작업을 필요로 한다. 이런 변환작업을 벡터화(vectorization)라고 하는데 효율적인 벡터화를 위해서는 컴퓨터가 자동으로 수행하는 변환 이외의 추가적 변환작업이 필요하게 된다[4, 9]. 현재 많이 연구되어 있는 벡터화 기법들은 주로 CRAY 및 IBM 계열의 컴퓨터와 같은 register-to-register 방식의 벡터컴퓨터를 대상으로 하고 있다[2, 3, 5]. 그러나 다른 벡터컴퓨터 중에는 register-to-register 방식과는 상이한 구조를 가진 것이 많으며 이런 구조적 차이점을 무시하고 register-to-register 방식에 적합하게 벡터화된 프로그램을 이식하여 사용할 경우에는 오히려 성능저하를 초래할 수있다[6].

본 논문에서는 register-to-register 방식과는 다른 memory-to-memory 방식의 벡터컴퓨터에서의 외연적 유한요소법의 효율적인 벡터화 방법을 제시하였다. 먼저 벡터컴퓨터의 구조적 특성과 무관하게 적용될 수 있는 일반적인 벡터화 기법을 고찰한 후 memory-to-memory 방식의 벡터컴퓨터에 알맞는 벡터화 기법을 개발하였다. 개발된 벡터화 기법의 유용성을 확인하기 위해 대표적인 외연적 유한요소 프로그램인 DYNA3D를 memory-to-memory 방식의 벡터컴퓨터인 HDS AS/XL V50에 이식하여 성능 향상을 확인하였다.

2. 벡터처리 및 벡터화

2.1 벡터처리

많은 공학분야에서 복잡한 문제의 해결을 위하여 보다 강력한 계산능력을 갖춘 컴퓨터가 요구되어 왔다. Von Neuman 방식의 컴퓨터로는 방대한 수치 데이타를 효과적이고 신속하게 처리하는 데 한계가 있기 때문에 병렬처리(parallel processing)라는 새로운 개념의 처리방식이 제시되었다. 벡터처리(vector processing)는 다중처리(multiple processing)와 함께 대표적인 병렬처리 방식이다[1]. 여기서 벡터란 연속적으로 처리될 수 있는 데이타들의 모임으로 의미가 다소 유동적이다[1, 6]. 다시말해 어떤 벡터처리기에서는 벡터로 취급될 수 있는 것이 다른 벡터처리기에서는 그렇지 못할 수도 있다. 연속된 배열(array)은 벡터의 대표적 예이다. 일정한 간격(stride)으로 떨어져 있는 배열은 대부분의 벡터처리기에서 벡터로 취급되나 CYBER 205와 같은 기계에서는 벡터로 처리되지 못한다.

벡터컴퓨터는 벡터처리기를 내장하여 벡터처리 기능을 갖춘 컴퓨터를 말하며 대개의 경우 스칼라처리기를 함께 갖고 있다[1]. 거의 모든 슈퍼컴퓨터는 벡터컴퓨터이며[1, 7], 기존의 스칼라컴퓨터에 추가적인 벡터처리기(attached vector processor)를 붙이는 경우도 있다[1, 8].

벡터처리기는 피연산자(operand) 쌍들을 연속적으로 연산장치(functional unit)에 투입하므로써 각 연산단계(stage)에서 서로 다른 피연산자 쌍들의 계산을 동시에 수행하도록 하는 pipelining을 사용함으로써 큰 성능향상을 얻는다[1. 9]. Pipelining 기법은 스칼라처리기에서도 이용되고 있으나 같은 연산이 계속 반복되는 벡터처리기에서 훨씬 크게 효과를 본다. 벡터처리의 다른 강점은

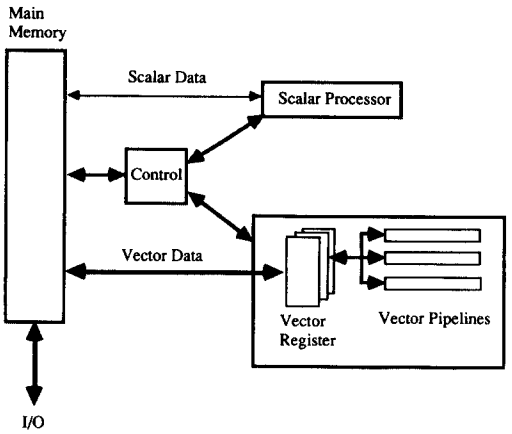

그림1. Register-to-Register Type Vector Computer

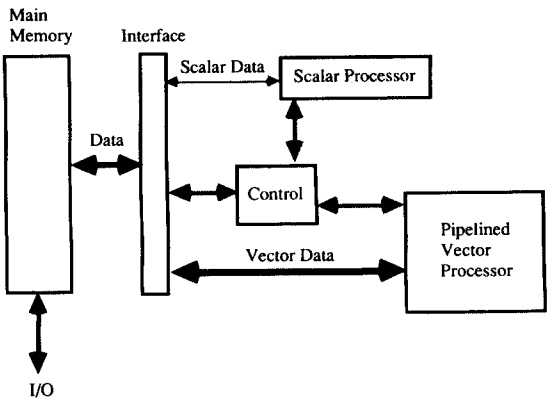

그림2. Memory-to-Memory Type Vector Computer

반복되는 연산을 단 한번의 명령어(instruction)를 사용하여 처리한다는 것이다. 이것은 반복되는 명령어와 loop 제어에 의한 손실을 줄여주므로써 벡터 데이타를 훨씬 빠르게 처리할 수 있게 해준다[1].

벡터 컴퓨터는 크게 두가지 종류로 나누어진다[1]. 하나는 register-to-register 방식으로 주기억장치와 연산장치 사이에 여러개의 벡터 register를 두어서 상대적으로 느린 주기억장치의 호출속도를 빠른 register로 보완한다(그림1). 이런 종류의 대표적인 컴퓨터로서 CRAY계열과 IBM 3090/VF가 있다. Register의 속도는 주기억장치의 속도에 비해 엄청나게 빠르므로 벡터 연산장치의 처리속도에 맞추어 데이타를 주고 받을 수 있다. 따라서 처리기안에서 빠른 처리속도를 제공한다. 그러나 register의 크기는 주기억장치의 용량에 비해 아주 작으므로 큰 벡터를 취급하는데는 sectioning overhead가 따른다[9, 10]. 다른하나는 memory-to-memory 방식으로 주기억장치와 연산장치 사이에 데이타 교환이 직접 이루어진다. CDC CYBER 205와 HDS AS/XL이 이런 종류의 컴퓨터이다(그림2). 주기억장치의 호출속도는 연산장치의 속도에 비해 현저히 느리므로 기억장치를 모듈화하거나 계층화하여 기억장치의 대역폭(bandwidth)을 늘려주어야 한다[7]. HDS AS/XL 과 같은 컴퓨터는 cache를 도입하여 속도차를 좁히고 있다[11]. 이런 종류의 컴퓨터는 긴 벡터의 처리에 우수한 성능을 발휘한다[9, 10].

2.2 벡터화

기존의 프로그램에서 벡터처리능력을 최대한 이용하기 위해서는 프로그램의 일부 또는 전부를 적합한 병렬처리언어(parallel language)로 재작성하는 것이다. 그러나 현재로서는 표준화된 병렬처리언어는 존재하지 않으며[1], 더우기 방대한 양의 기존 프로그램을 거의 전부 재작성하는 것은 새로 프로그램을 작성하는 것보다 더 많은 노력과 비용을 요구한다. 결국 FORTRAN과 같은 기존의 순차처리언어(sequential language)로 쓰여진 프로그램을 컴퓨터가 벡터처리에 맞게끔 자동으로 변환해주는 것이 대안으로 제시되었다. 이 방법은 매우 적은 노력으로 기존의 프로그램을 벡터처리할 수 있다는 장점이 있다. 그러나 컴퓨터의 자동변환작업은 algorithm 자체를 변환해 주지는 못하므로 한계가 있으며 벡터처리 능력을 최대한 이용하기 위해서는 추가적인 수정작업이 필요하다[4, 9]. 이런 추가적인 수정작업을 포함하여 기존의 순차처리언어로 작성된 프로그램을 벡터처리가 가능하도록 바꾸는 모든 작업을 합하여 벡터화(vectorization)라고 부른다. 또 기존의 프로그램을 벡터처리가 가능하도록 자동으로 변환해주는 소프트웨어를 벡터변환기(vectorizer, vector compiler)

라고 한다.

FORTRAN으로 쓰여진 프로그램에서 벡터처리가 가능한 부분은 DO－loop 및 DO－loop로 변환이 가능한 loop 구조이다[1, 9]. 이때 DO－loop 안의 실행문들은 loop 순서에 의존적이지 말아야 한다. 벡터변환기는 DO－loop 중 벡터처리가 가능한 것들을 탐색하고 분석하여 벡터처리 명령어로 구성된 모듈들로 대치한다. 이때 일반적인 스칼라 부분의 처리는 기존의 컴파일러가 담당한다.

추가적인 수정작업은 자동변환작업을 돕기위해 약간의 문법적 수정 및 재배치를 행하는 단순한 작업으로부터 전체적인 algorithm을 재구성하는 작업까지 여러단계가 있을 수 있다. 복잡한 수정작업은 더 많은 성능향상을 가져올 수 있지만 더 많은 노력과 비용이 든다. 따라서 벡터화 목표에 따라 적절히 수정작업의 단계를 조절할 필요가 있다.

3. 외연적 유한요소법의 벡터처리

내연적 유한요소법은 해의 절대적 안정성(unconditionally stable)에도 불구하고 동적비선형 문제에서는 엄청난 계산시간과 기억용량이 요구된다는 문제점이 있다. 이런 점은 3차원 문제에서는 더욱 심화되어 충돌해석과 같이 많은 시간간격(time step)을 갖는 문제에서는 너무 많은 계산비용이 소요된다. 이런 점에서 3차원 동적 비선형 문제를 외연적 방법으로 해결하려는 것은 매우 당연하다 하겠다.

외연적 유한요소법은 내연적 방법에 비해 빠른 수행속도를 보장하며 요구되는 기억용량도 훨씬 적다[3]. 또한 벡터처리에도 매우 적합한 구조를 갖고 있어 벡터컴퓨터에서는 더욱 빠르게 수행될 수 있다. 시간 t^n에서의 가속도는 다음식으로 구해진다.

$$\boldsymbol{M}\boldsymbol{a}^n=\boldsymbol{P}^n-\boldsymbol{F}^n \tag{1}$$

여기서, $\boldsymbol{M}$: 질량 매트릭스
$\boldsymbol{a}^n$: t^n에서의 가속도
$\boldsymbol{P}^n$: t^n에서의 외부 힘 벡터
$\boldsymbol{F}^n$: t^n에서의 내부 힘 벡터

질량 매트릭스 $\boldsymbol{M}$을 "Lump Mass Matrix"형태로 계산하면 가속도 벡터 $\boldsymbol{a}^n$의 성분 a_i를 직접 구할 수 있다. 즉,

$$a_i^n=(P_i^n-F_i^n)/M_i \tag{2}$$

속도벡터 $\boldsymbol{v}$와 변위벡터 $\boldsymbol{x}$는 중간차분법(central difference method)에 의해 구해진다.

$$\boldsymbol{v}^{n+\frac{1}{2}}=\boldsymbol{v}^{n-\frac{1}{2}}+\boldsymbol{a}^n\Delta t^n \tag{3}$$

$$\boldsymbol{x}^{n+1}=\boldsymbol{x}^n+\boldsymbol{v}^{n+\frac{1}{2}}\Delta t^{n+\frac{1}{2}} \tag{4}$$

여기서, $\Delta t^{n+\frac{1}{2}}=(\Delta t^n+\Delta t^{n+1})/2$ (5)

전체계산에서 내부응력에 의한 힘 $\boldsymbol{F}^n$을 구하는 과정에 가장 많은 계산량을 소요하게 된다.

$$\boldsymbol{F}^n=\sum_e \int_{v_e} \boldsymbol{B}^T\sigma^n dv \tag{6}$$

여기서, $\boldsymbol{B}$: 변형도－변위 매트릭스
σ^n : t^n에서의 응력벡터
v_e : 요소의 체적

이 계산은 주로 요소(element) 단위에서 진행되며 마지막에 각 요소에서 계산된 힘은 각 절점으로 합산된다(이하 이 계산부분을 "요소처리부분"이라 한다). 이렇게 요소 단위의 힘을 각 절점으로 합산시키는 과정을 Right－Hand Side(RHS) assembly라고 한다. 외연적 유한요소법의 전체계산에서 $\boldsymbol{F}^n$을 구하는 과정이 차지하는 비율은 85% 이상이다.

그림3은 일반적인 외연적 유한요소법의 계산과정을 나타내고 있다. 요소단위의 계산에서는 각 요소들 사이의 정보교환이 필요없으므로 완전히 독립적으로 수행될 수 있다. 또한 속도와 변위를 구하는 과정(식(3), 식(4))에서도 각 절점의 계산은 다른 절점의 계산과 무관하다. 결국 외연적 유한요소법에서는 대부분의 계산이 독립적으로 수행될 수 있다. 많은 양의 독립적인 정보들의 연산을

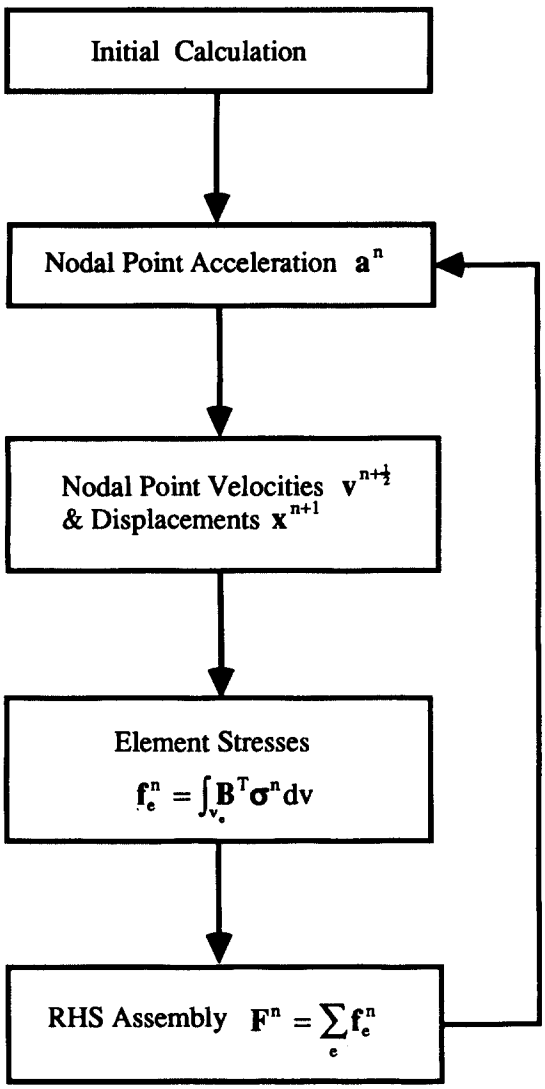

그림3. Flow Diagram of the Explicit Finite Element Method

동시에 수행하는 것이 벡터처리이므로 외연적 유한요소법은 벡터처리에 매우 적합하다. 단, 마지막 계산인 RHS과정에서 데이타간의 의존성(dependency)이 존재하며 일반적인 방법으로는 벡터화되지 않는다[12]. 그러나 이 과정도 특수한 algorithm을 도입하여 벡터화시키는 노력들이 진행중이며 본 논문에서도 새로운 algorithm이 제안되어 있다.

4. 일반적 벡터화 기법

3장에서 기술한 바와 같이 스칼라컴퓨터에선 요소처리부분의 계산은 각 요소별로 순차적으로 수행된다. 원래 순차처리용 algorithm에서는 다음과 같은 loop가 하나 이상 존재한다.

1) 각 요소(element)에 대한 loop
2) 각 자유도(degree of freedom)에 대한 loop
3) 각 요소절점(element node)에 대한 loop
4) 각 적분점(integration point)에 대한 loop

4)와 같은 loop는 one-point 적분으로 제거될 수 있다. 또 2), 3)과 같은 loop도 계산 속도의 증대를 위해 펼쳐진다(unscrolling). 이와 같은 기법은 스칼라와 벡터컴퓨터 모두에서 효과적이다[10, 12]. 1)번 loop는 스칼라의 경우 기억장치를 효율적으로 사용하기 위하여 요소처리부분의 맨 바깥에서 운용된다. 즉,

```
DO 100 I=1, NUMEL
      .
      .
      .
F11=SIG1*PRR1+SIF4*PZZ1
F21=SIG1*PRR1+SIF4*PZZ1
F12=SIG1*PRR2+SIF4*PZZ2
F22=SIG2*PRR2+SIF4*PZZ2
      .
      .
      .
RHS ASSEMBLY PART
100 CONTINUE
```

(A)

(A)와 같은 구조에서는 한번의 loop에 한요소씩 계산이 된다. 벡터화를 위해서는 각 스칼라를 배열(array)로 치환시켜 기능별로 DO-loop에 넣는다[3]. 즉,

```
DO 100 I=1, NUMEL
      .
      .
      .
```

```
100 CONTINUE
      .
      .
      .
   DO 200 I=1, NUMEL
   F11(I)=SIG1(I)*PRR1(I)+SIG4(I)*PZZ1(I)
   F21(I)=SIG2(I)*PZZ1(I)+SIG4(I)*PRR1(I)
   F12(I)=SIG1(I)*PRR2(I)+SIG4(I)*PZZ2(I)      (B)
   F22(I)=SIG2(I)*PZZ2(I)+SIG4(I)*PZZ2(I)
200 CONTINUE
      .
      .
      .
 RHS ASSEMBLY PART
```

(B)의 DO－loop들은 벡터변환기에 의해 벡터처리가 가능하도록 변환되므로 결국 모든 요소의 계산이 동시에 진행된다. RHS assembly 부분은 단순히 벡터화되지 않는 scatter 연산들로 이루어져 있다. (B)와 같은 코드는 벡터화가 가능하지만 요소의 갯수(NUMEL)가 매우 많을 때는 벡터길이(vector length)가 너무 길어져 더이상 벡터처리 속도가 향상되지 않거나 오히려 저하되는 문제가 발생한다. 더우기 (B)와 같은 구조에서는 요소의 갯수에 따라 배열의 크기가 변화하므로 기억용량의 설정에도 문제가 있다.

이와 같은 문제를 해결하기 위해서는 요소와 관련된 배열들을 적당한 크기로 그룹(group)화할 필요가 있다[3]. 결국 분할된 그룹의 크기는 벡터길이가 되며, 그룹크기의 설정이 전체 수행성능에 큰 영향을 미친다. NUMGRP개의 그룹으로 분할된 후의 DO－loop는 다음과 같다.

```
DO 10 IGROUP=1, NUMGRP
LFT=IGROUP번째 그룹의 첫 요소번호
LLT=IGROUP번째 그룹의 마지막 요소번호

DO 100 I=1, NUMEL
      .
      .
      .
100 CONTINUE
      .
      .
      .
   DO 200 I=LFT, LLT
   F11(I)=SIG1(I)*PRR1(I)+SIG4(I)*PZZ1(I)
   F21(I)=SIG2(I)
   F12(I)=SIG1(I)*PRR2(I)+SIG4(I)*PZZ2(I)      (C)
   F22(I)=SIG2(I)*PZZ2(I)+SIG4(I)*PRR2(I)
200 CONTINUE
      .
      .
      .
 RHS ASSEMBLY PART

10 CONTINUE
```

여기서 LFT와 LLT는 각 그룹의 경계를 나타낸다.

(LLT－LFT＋1)의 그룹크기가 되며 설정된 최대그룹크기를 넘지 않도록 조절된다.

5. 컴퓨터 특성을 고려한 벡터화 기법

지금까지 기술한 벡터화 기법은 일반적인 것으로서 벡터컴퓨터의 architecture의 특성과는 관계없이 적용될 수 있다. 그러나 보다 효과적인 벡터화를 위해서는 벡터컴퓨터의 특성을 고려하여야만 한다. 지금까지는 CARY와 같은 register－to－register 방식의 컴퓨터에서의 벡터화에 관한 연구가 많이 진행되었는 바, 다른 architecture의 벡터컴퓨터에서도 거의 그대로 이방식을 위하여 작성된 프로그램을 사용하여 왔다. 이 경우 비록 자동벡터변환기에 의해 특정 기종에서 사용가능하도록 벡터화가 이루어지지만 특정 벡터처리기의 성능을 최대한으로 이용한다고 볼 수는 없다. Memory－to－memory 방식의 벡터컴퓨터는 register－ to － register 방식과는 매우 다른 architecture를 가지고 있다. 최근의 memory－to－memory 방식에서는 보다 높은 meomory band-

width를 제공하기 위해 기억장치를 계층화시켰고 특히 interface로서 cache를 사용하는 경우도 있다. Cache는 register-to-register 방식에서도 사용되는 경우가 있으며[13] 어떤 방식에서든 전체 성능에 큰 영향을 미친다[7, 10]. 본 장에서는 cache를 갖고 있는 memory-to-memory 방식의 벡터컴퓨터에서의 벡터화 기법을 기술한다.

앞에서 언급한 바와 같이 요소 데이타의 그룹화가 필요하다. 이때 이 그룹의 크기(LLT-LFT+1)가 벡터길이가 되므로 전체성능에 큰 영향을 미치게 된다. 보통 다음과 같은 것이 그룹크기의 결정시 고려되어야 한다.

1) Memory-to-memory 방식의 벡터처리기는 긴 벡터의 처리에 적합하다. 만약 아주 짧은 벡터를 벡터처리할 경우는 스칼라로 처리하는 것보다 더 낮은 성능을 보인다[1, 10]. 이 경계점을 break-even point라고 하는데 각 기종마다 상이하며 일반적으로 memory-to-memory 방식이 register-to-register 방식보다 더 긴 벡터를 요구한다[10].

2) Cache가 있을 경우 벡터 길이가 cache의 용량을 초과하면 cache의 적중률(hit ratio)이 감소되어 실행성능이 떨어진다[10, 13]. Cache의 적중률을 높이기 위해서는 여러번 전의 명령에서 쓰였던 데이타들도 cache에 저장되어있어야 하므로 긴 배열이 사용되면 매우 불리하다. Cache에 저장되는 내용은 프로그램에서 조정할 수 없으며 cache 조정기(controller)가 자동으로 해주는데 보통 LRU(Least Recently Used : 가장 최근에 쓰이지 않은 것들부터 대치된다) algorithm이 사용된다[1, 7].

3) 그룹크기의 증가는 그에 따른 여러 배열들의 크기(dimension)를 증가시켜 많은 기억공간을 요구하게 된다. 이때 가상기억시스템(virtual memory system)에서는 잦은 paging이 요구되어 실행성능을 떨어뜨린다.

Cache가 없을 경우는 1)과 3)에 의해서, cache가 있을 경우는 1)과 2)에 의해서 최적의 그룹크기를 결정한다.

DYNA3D의 경우 원래 CRAY-1을 위하여 작성되었기 때문에 그룹크기를 128로 하였다[3, 14]. CRAY는 64 element 크기의 register를 갖고 있다. 따라서 64의 배수중에 적당한 값을 선택했는데 CRAY-1의 주기억장치크기가 작기 때문에 (1M Words) 128로 결정되었다. 지금과 같이 CRAY 컴퓨터의 용량이 커졌을 때는 좀 더 큰 값을 택하는 것이 효율적일 것이다.

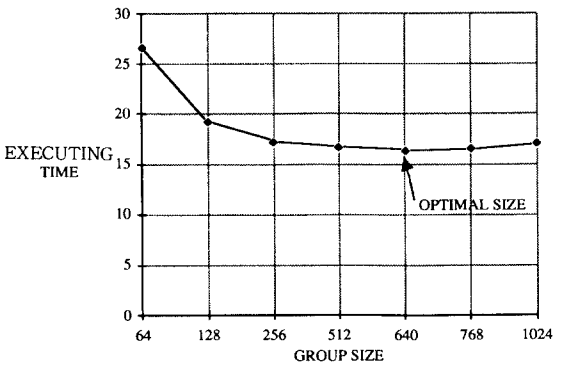

그림4. Effect of the Group Size on Performance

DYNA3D 프로그램을 다른 방식의 벡터컴퓨터에 적합하게 고칠 경우에도 128보다 큰 값이 채택되어야 한다. 실제로 cache가 있는 memory-to-memory 방식의 벡터컴퓨터인 HDS AS/XL V50에서 실험해 본 결과 640 4byte-Words가 최적의 그룹크기로 나타났다(그림4). AS/XL V50의 break-even point는 약 20이며 cache의 크기는 128KB이다. 단일 이항연산에서 최대로 cache에 들어갈 수 있는 벡터쌍의 길이는 8,000 4byte-Words이다. 그러나 실제 프로그램에서는 많은 배열이 cache에 존재할수록 적중률이 올라가므로 훨씬 작은 값에서 최적값이 나타난다. 그림4는 그룹크기가 전체 성능에 얼마나 큰 영향을 미치는가를 보여준다. 최적그룹크기를 결정짓는 중요 요소는 다음과 같다.

1) 프로그램에서 계산시간을 많이 소모하는 부분의 algorithm
2) 프로그램의 실수, 정수 데이타형이 차지하는 기억크기
3) cache의 크기와 architecture

최적 그룹크기를 결정하는 방법은 중요 부분을 성능분석장치(performance analysis tool)로 추적하여 조정하는 것이다. 성능분석장치가 없을 경우의 가장 간단한 방법은 그림4와 같이 그룹크기를 변화시켜 최적값을 구하는 것이다. 그러나 이 경우는 응용예제별로 최적값이 다를 수 있다는 단점이 있다.

벡터를 계산하는 과정에서는 gather와 scatter 연산들이 포함된다. gather연산은 요소처리부분의 맨앞에 나타나는데 절점의 속도와 변위 데이타를 각 요소절점으로 할당시키는 역할을 한다. 이 연산은 초기의 벡터컴퓨터에서는 벡터처리가 불가능했으나 최근에는 가능해졌다. 다음은 gather 연산의 예다.

```
DO 100 I = LFT, LLT
  VX(I) = V(1, IX1(I))
  VY1(I) = V(2, IX1(I))                    (D)
  VZ1(I) = V(3, IX1(I))
100 CONTINUE
```

여기서 IX1(I)는 I번째 요소의 첫번째 요소절점의 절점번호, V(1, IX1(I))는 IX1(I)번째 절점의 첫번째 자유도 방향으로의 속도성분을 나타낸다. (D)의 구조는 DYNA3D와 같은 기존의 프로그램에서 사용하는 것인데 벡터처리에는 부적당하다. FORTRAN의 배열구조는 열(column) 우선 저장방식으로 되어있다. 그런데 벡터처리시 VX1(I) 배열의 연산이 모두 끝난후 VY1(I)배열의 연산이 수행되므로 V배열은 3의 간격(stride)을 갖는 벡터로 취급된다. 벡터처리는 간격이 1일때 가장 효율적이다[5, 7].

또한 (D)의 구조는 cache의 운용면에서도 매우 불리하다. 즉 cache에는 line이라는 block 단위로 주기억장치의 내용이 복사되므로 어떤 데이타가 한번 호출되면 주변 데이타들도 cache에 함께 저장된다. 이때 한 번의 벡터연산에서 필요한 데이타들은 V배열에 3의 간격으로 떨어져 있으므로 cache에는 당장 필요없는 데이타들이 많아져 적중률을 떨어뜨린다(그림5).

효율적으로 gather operation을 운용하기 위해서

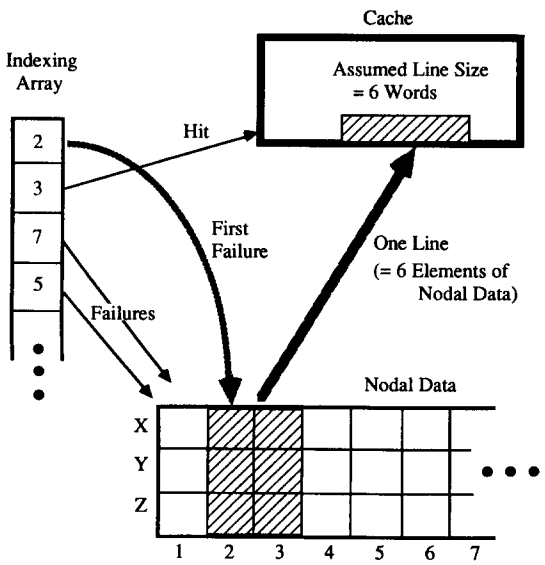

그림5. Nodal Data Structure and Cache

는 V와 X를 각각의 자유도 별로 나누어 1의 간격을 갖는 벡터 VX, VY, VZ로 변환시켜준 후 (D)를 다음과 같은 algorithm으로 수정한다.

```
DO 100 I = LFT, LLT
  VX1(1) = VX(IX(I))
  VY1(I) = VY(IX1(I))                      (E)
  VZ1(I) = VZ(IX1(I))
100 CONTINUME
```

절점의 속도 및 변위 벡터는 프로그램 전체에서 사용되기 때문에 전체적인 개조에는 많은 비용과 노력이 든다. 따라서 gather operation이 있는 요소처리부분에서만 배열 구조를 변환해 주는 것이 간편하게 효과를 얻을 수 있는 방법이다. 이때 매번 배열구조를 변환해주는 overhead가 생기는데 절점데이타 구조의 개조로 생기는 이익의 약 25%를 차지한다. DYNA3D를 통한 실험결과 전체적으로 5%의 성능향상을 나타냈으며 부분개조가 아닌 전체개조 시에는 좀 더 많은 성능향상을 보일것으로 예상된다.

6. RHS assembly 부분의 벡터화

요소처리부분중 마지막으로 힘을 합산하는

RHS assembly 부분은 일반적인 방법으로는 벡터화가 되지 않는다. 다음은 전형적인 RHS assembly 부분의 예이다.

```
DO 100 I LFT, LLT
  RHS(1, IX1(I)) = RHS(1, IX1(I)) + EP11(I)
  RHS(2, IX1(I)) = RHS(2, IX1(I)) + EP21(I)
  RHS(3, IX1(I)) = RHS(3, IX1(I)) + EP21(I)
  RHS(1, IX2(I)) = RHS(1, IX2(I)) + EP12(I)
  RHS(2, IX2(I)) = RHS(2, IX2(I)) + EP22(I)
  RHS(3, IX2(I)) = RHS(3, IX2(I)) + EP32(I)
  RHS(1, IX3(I)) = RHS(1, IX3(I)) + EP13(I)        (F)
  RHS(2, IX3(I)) = RHS(2, IX3(I)) + EP23(I)
  RHS(3, IX3(I)) = RHS(3, IX3(I)) + EP33(I)
  RHS(1, IX4(I)) = RHS(1, IX4(I)) + EP14(I)
  RHS(2, IX4(I)) = RHS(2, IX4(I)) + EP24(I)
  RHS(3, IX4(I)) = RHS(3, IX4(I)) + EP34(I)
100 CONTINUE
```

여기서 RHS는 F^n벡터를 나타내는 배열이며, IX1, IX2, IX3, IX4은 각 요소절점의 전체절점들과의 connectivity를 나타내는 배열, EP11, EP21, EP31 등은 요소의 내력들이다.

RHS 배열은 IX1 등에 의해 간접참조(indirect indexing)가 되는데 이때 IX1 등이 중복되는 값을 갖게되는 경우에는 벡터처리시에 스칼라처리와는 다른 결과를 가져오게되어 벡터화가 불가능하다. 이 부분을 벡터화하기 위해서는 특수한 algorithm이 필요한데 Ferentz의 "Coloring Algorithm"이 대표적이다[2, 12]. Ferentz는 (F)의 연산이 그룹단위로 이루어진다는 데 착안하여 한 그룹안에서는 서로 인접하지 않는 요소들만 포함시키도록 재배열하므로써 IX1 등이 중복되는 값을 갖지 않게하여 벡터화가 가능하도록 하였다. 이때 인접하지 않는 요소들끼리 그룹화하기 위해 인접 각 요소를 색칠하되 인접요소간에는 같은 색깔이 되지 않도록 한다는 기본 개념을 도입하여 algorithm을 구현하였다. 원래 이 개념은 내연적 유한요소법의 벡터처리를 위해 개발되었으며 외연적 유한요소법에서도 적용되고 있다고 한다. Ferentz의 algorithm은 간단명료한 개념임에도 불구하고 사면체(tetrahedral) 및 오면체(prismatic) 요소들이 포함되는 경우에는 구현하기가 힘들다는 단점이 있다[12].

본 연구에서는 Ferentz와는 다른 시각에서 RHS 부분의 벡터화를 수행하였다. 먼저 (F)를 자세히 고찰하여 다음과 같은 사실을 발견하였다.

"RHS 부분의 각 연산은 요소별로 진행된다기보다는 각요소의 요소절점별로 계산이 수행되며, 따라서 비록 인접한 요소들이 한 그룹에 있어도 요소절점번호가 중첩되지 않으면 동시에 계산될 수 있다."

예를 들어 그림6과 같은 구조를 갖는 유한요소망을 생각하자. 표1(a)의 요소번호 구조를 갖는 경우 IX1(I)는 1번과 2번 요소가 IX4(I)는 2번과 4번요소가 중첩되어 벡터처리시 오류를 발생시킨다. 그러나 요소번호구조를 표1(b)와 같은 구조로 갖는다면 어떤 배열도 중첩되는 요소가 없게되어 벡터처리를 수행해도 스칼라 처리와 같은 결과를

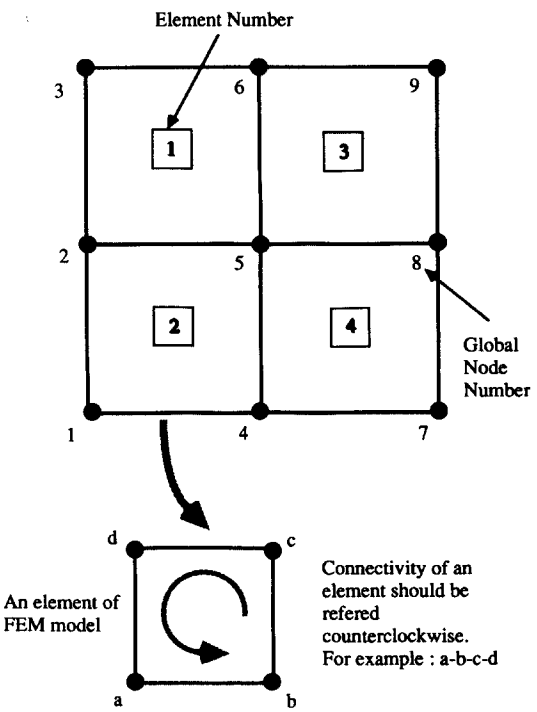

그림6. Topology of the Example Model

표1. Node Connectivities of the Example Model

(a) Dependent case

Element Number	Node Connectivity			
	IX1	IX2	IX3	IX4
1	5	6	3	2
2	5	2	1	4
3	8	9	6	5
4	7	8	5	4

(b) Independent case

Element Number	Node Connectivity			
	IX1	IX2	IX3	IX4
1	5	6	3	2
2	4	5	2	1
3	8	9	6	5
4	7	8	5	4

얻는다. 이제 간단한 예를 들어RHS 부분의 벡터화가 불가능한 이유를 고찰해 보자. 이때 배열 IX가 {1, 2, 1, 3}과 같은 배열성분을 가지고 있어 첫번째와 세번째 index 값이 중복된다고 가정한다.

```
DO 100 I=1, 4
100 RHS(IX(I))=RHS(IX(I))+F(I)        (G)
```

(G)를 기존의 스칼라컴퓨터에서 계산하면 다음과 같은 순서로 수행된다.

```
RHS(1)⇐RHS(1)+F(1)
RHS(2)⇐RHS(2)+F(2)
RHS(1)⇐RHS(1)+F(3)                    (H)
RHS(3)⇐RHS(3)+F(4)
```

여기서 RHS(1)을 살펴보면 다음과 같은 식과 동치이다.

$$RHS(1)=RHS(1)+F(1)+F(3) \qquad (I)$$

그러나 (G)를 벡터처리하면 그림7과 같은 방식으로 연산이 된다. 이때 RHS(1)의 결과는 X(1) 배열의 값이 저장되고 다시 X(3) 배열에 의해 중첩된다. 즉,

$$RHS(1)=RHS(1)+F(3) \qquad (J)$$

(J)와 (I)는 다르며 이는 index의 중복에 의해 발생된다. 그러나 RHS(1)을 제외한 나머지 성분의

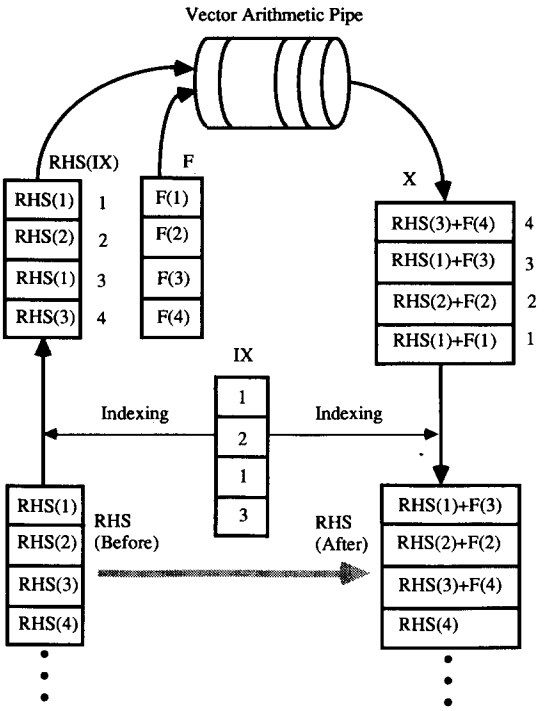

그림7. Vector Processing of the RHS Assembly Part

계산에는 아무런 오류가 없다. 또한 실제의 모델에서 같은 그룹내의 요소들이 동일한 요소번호의 공통절점을 갖는 경우는 매우 드물다. 따라서 이런 특수한 경우에 해당하는 요소절점들만을 제외한 후 나머지 대부분의 요소절점에 대해서는 벡터화를 진행해도 오류가 발생하지 않는다. 이런 점에 착안하여 그림8과 같은 algorithm을 개발했다. 이 algorithm에 따라 (G)를 계산하면 :

(1) 준비단계에서 IX배열에 중복되는 번호1이 있음을 찾아둔다.

(2) DUMP(1)⇐F(1)
DUMP(2)⇐F(3)
F(1)⇐0.
F(3)⇐0.

(3) X(1)⇐RHS(1)+0.
X(2)⇐RHS(2)+F(2)
X(3)⇐RHS(1)+0.
X(4)⇐RHS(1)+0.
RHS(1)⇐X(1)

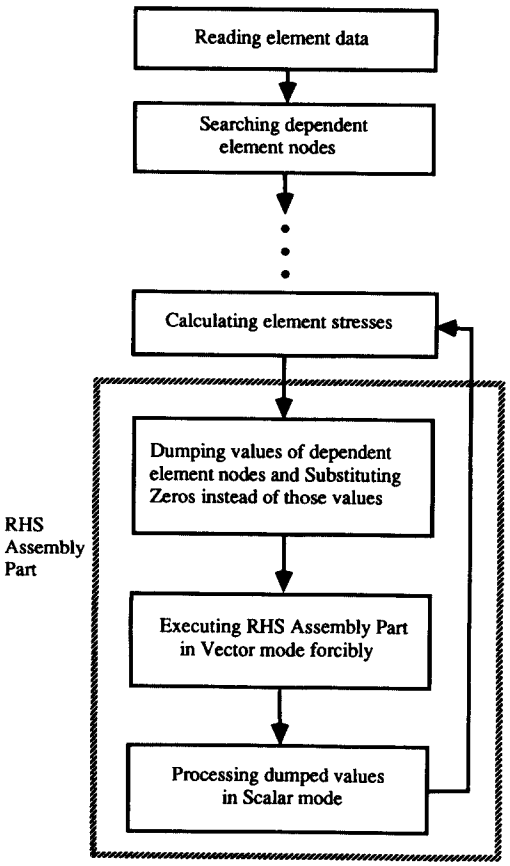

그림8. Vectorization Algorithm of the RHS Assembly Part

RHS(2)⇐X(2)

RHS(1)⇐X(3)

RHS(3)⇐X(4)

결과 :

RHS(1) = RHS(1)

RHS(2) = RHS(2) + F(2)

RHS(3) = RHS(3) + F(4)

(4) RHS(1)⇐RHS(1) + DUMP(1) + DUMP(2)

결과 :

RHS(1) = RHS(1) + F(1) + F(3)

결국 스칼라방식으로 처리한 것과 같은 결과를 얻게 된다.

새로운 algorithm을 DYNA3D에 적용하여 보았다. AS/XL상에서 RHS assembly부분은 전체의 약 10%를 차지한다. 새로운 algorithm은 기존의 순차적 algorithm보다 2배정도 빨리 수행되었으며 전체적으로 볼때 약 5% 정도의 성능향상을 가져온다.

7. 벡터화 예

본 논문에서 기술한 벡터화 기법을 대표적인 외연적 유한요소 프로그램인 DYNA3D에 적용시켜 보았다.

DYNA3D는 1976년 LLNL(Lawrence Livermore National Laboratory)에서 처음 개발되었으며 1979년 CRAY-1의 벡터기능을 이용하기 위해 벡터화되었다. 이때 4장의 처음부분에서 기술한 일반적인 벡터화기법을 적용하였으며 CRAY의 고속 계산능력을 이용하여 실행효율을 몇배나 증가시켰다[3, 14]. 그후 내부적인 수정을 거쳐 최근까지 널리 사용되고 있다. DYNA3D는 특히 충돌문제의 해석에 매우 우수한 성능을 발휘한다 최근에도 DYNA3D에 대한 개량이 진행되고 있으며 여러 벡터컴퓨터에 이식되어 사용되고 있다.

DYNA3D의 추가적인 벡터화 작업을 수행한 컴퓨터는 HDS의 AS/XL V50으로 memory-to-memory 방식을 취하며 256 KB의 HSB라는 cache를 내장하고 있다. AS/XL은 IBM 3090과 완전 호환으로 운영되나 벡터처리 architecture는 상이한 구조를 갖는다.

본 논문에서 기술한 벡터화 기법으로 대부분이 효과적으로 벡터화될 수 있었다. 이때 지수가 실수인 연산(Y(I) = A(I)**X)은 벡터화되지 못하는데 약간의 변환으로 쉽게 벡터화시킬 수 있다. 즉,

$$A^x = e^{\log_e A^x} = e^{x \log_e A} \tag{7}$$

Log와 Exp함수는 벡터처리가 가능하므로 3번에 나눠 계산하면 된다.

Y(I) = ALOG(A(I))

Y(I) = X*Y(I)

Y(I) = EXP(Y(I))

DYNA3D에서는 그룹으로 나누어진 요소들은

물질모델(material model) 종류별로 다시 그룹화된다. 즉 한 그룹안에 서로 다른 물질모델이 있으면 동시에 처리될 수 없으므로 소그룹으로 나눠지게 된다. 보통 유한요소망 생성시 물질모델별로 모아서 numbering을 하지는 않으므로, 경우에 따라서는 매우 작은 소그룹으로 나눠질 수도 있다. 이 경우 벡터길이가 작아지게 되며, 벡터처리 효율에 치명적인 손실을 가져온다. 이런 손실을 방지하기 위해서 본 연구에서는 요소 데이타 입력직후 같은 물질모델별로 요소 데이타를 모으는 간단한 algorithm을 고안하였다.

표2. Description of the Test Examples

Data	Description	Node	Element	Time*
EX1	DYNA3D finite element option applied to NIKE2D test problem	1377	Solid:973	0.5E+02
EX2	DYNA3D cylinder drop calculation	4808	Solid:3433	0.2E-03
EX3	Square cross section for single surface contact test	1911	4-Node Shell: 1800	0.5E-03

Note/* Final Destination Time of the Solution

표3. Performace Results in Terms of CPU Time

Unit : Sec

DATA	SCALAR		VECTOR1		VECTOR2	
	TOTAL	KERNEL	TOTAL	KERNEL	TOTAL	KERNEL
EX1	271.8	258.1	141.1	132.7	108.6	100.6
EX2	236.3	223.9	179.9	171.6	89.71	81.62
EX3	230.7	204.8	116.0	101.6	96.53	82.70

표4. Performance Ratios

DATA	VECTOR1/SCALAR		VECTOR2/VECTOR1		VECTOR2/SCALAR	
	TOTAL	KERNEL	TOTAL	KERNEL	TOTAL	KERNEL
EX1	1.93	1.94	1.30	1.32	2.50	2.57
EX2	1.31	1.30	2.01	2.10	2.63	2.74
EX3	1.99	2.02	1.20	1.23	2.39	2.48

AS/XL에서 벡터화된 DYNA3D의 성능향상을 알아보기 위해 표2와 같은 예제들로 시험해 보았다. 예제1과 예제2는 solid 요소로 이루어져 있고 예제3은 shell 요소로 되어 있다. 표3은 각 예제들의 수행결과가 나타나 있다. 여기서 SCALAR는 DYNA3D를 스칼라 방식으로 수행했을때의 결과, VECTOR1은 CRAY에서의 벡터화를 위해 작성된 프로그램을 AS/XL에서 벡터화했을 때의 결과, VECTOR2는 AS/XL의 벡터처리 특성에 맞게 추가적인 수정작업을 해준 프로그램의 벡터처리 결과를 각각 소요된 CPU 시간별로 나타내고 있다. 표4는 표3에서 나타난 결과를 SCALAR, VECTOR1, VECTOR2들간의 비율로 나타내준다. 여기서 KERNEL이란 요소처리부분을 의미한다.

DYNA3D는 일반적인 벡터화 기법이 매우 잘 적용되어있기 때문에 다른 기종에서도 비교적 우수한 성능을 보였다. VECTOR1이 이런점을 나타내주는데 예제1과 예제3에서 VECTOR/SCALAR 비가 2.0에 이르고 있다. 예제2만이 30% 밖에 향상되지 못했는데 예제2의 요소 순서가 물질모델이 빈번하게 바뀌는 방식으로 배열되어 있기 때문이다.

추가적인 벡터화는 더욱 향상된 성능을 발휘하게 해준다. VECTOR2/VECTOR1은 추가적인 벡터화로 인해 성능이 20% 이상 향상됨을 보여주고 있다. 특히 예제2는 요소순서를 간단히 재배열하므로써 얼마나 큰 이득을 얻을 수 있는 가를 보여준다. 시험결과 재배열만으로도 45%의 성능향상을 보였다. 추가적인 벡터화는 기존 프로그램의 실행시보다 많은 기억용량을 요구하는데 전체적인 기억용량에서 차지하는 비중은 크지 않다. 또한 최근의 경향은 계산자원의 이용에서 제약으로 작용하는 것이 기억장치의 용량이 아니라 CPU의 계산속도이므로 계산속도의 향상을 위한 추가적인 기억용량의 사용은 크게 문제가 되지 않는다. 벡터화에 의한 전체적인 성능향상은 약 240% 정도이다.

8. 결론

본 논문에서는 벡터처리기의 특성을 고려한 외연적 유한요소법의 벡터화 방안을 모색하였다. 먼저 기존의 벡터화 기법중 벡터처리기의 특성과 관계없이 적용될 수 있는 일반적인 방법들을 체계적으로 정리해 보았고, 기존에 연구가 되어 있는 register-to-register 방식과는 다른 memory-to-memory 방식의 벡터컴퓨터에 알맞는 벡터화

기법을 개발하였다.

특히 cache가 있을 경우는 cache의 특성이 전체적인 성능에 큰 영향을 미치며, 효과적으로 cache를 운용하기 위하여서는 요소의 그룹크기를 최적화하고 절점과 관련된 데이타 구조를 cache의 적중률이 올라가도록 변환하여야 함을 알았다.

일반적인 방법으로 벡터화가 불가능한 Right-Hand Side assembly 부분의 벡터화를 위하여 새로운 algorithm을 개발하였다. 기존의 Ferentz의 algorithm이 요소단위의 관점에서 접근한 반면 본 algorithm은 각 요소의 절점단위에서 벡터화를 진행한다. 실제 예제를 통해 시험한 결과 안전하게 algorithm이 수행되며 RHS assembly 부분은 2배정도의 속도 향상을 보였다. 새로운 algorithm은 벡터처리기의 종류와 관계없이 적용될 수 있으므로 CRAY와 같은 register-to-register 방식의 컴퓨터에서도 큰 효과를 볼 것이다.

본 연구에서 개발한 여러 벡터화 기법의 유용성을 확인하기 위해 register-to-register 방식의 벡터화를 위해 작성된 외연적 유한요소해석 프로그램인 DYNA3D를 memory-to-memory 방식의 벡터컴퓨터인 HDS AS/XL V50으로 이식하였다. 이식결과 기본적인 벡터화 기법에 의해서도 벡터화가 잘 수행되어 스칼라에 비해 2배정도의 성능향상을 보였으며, 기종의 특성을 고려한 추가적 벡터화에 의해서도 20% 이상의 추가적인 성능향상을 보였다.

전체적으로 벡터화에 의해 2.4배 이상의 성능향상을 얻을 수 있었으며 이는 효과적인 벡터화의 필요성 및 유용성을 나타내준다.

감사의 글

본 논문의 일부는 미국 HDS사의 연구비 지원으로 수행되었으며, 이에 HDS사에 감사를 드립니다.

참 고 문 헌

1. K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company(1985).
2. T.J.R. Hughes and R.M. Ferencz, "Large-scale vectorized implicit calculations in solid mechanics on a CRAY X-MP/48 utilizing EBE preconditioned conjugate gradients," Comput. Meths. Appl. Mech. Engrg. 61, 215-248(1987).
3. G.L. Goudreau and J.O. Hallquist, "Recent Developments in Large-Scale Finite Element Lagrangian Hydrocode Technology," Comput. Meths. Appl. Mech. Engrg. 33, 725-757(1982).
4. R.D. Vanluchene, R.H. Lee and V.J. Meyers, "Large scale finite element analyses on a vector processor," Computers & Structures 24, 625-635(1986).
5. P. Angeleri, D.F. Lozupone, F. Piccolo and J. Clinckemaillie, "PAM-CRASH on the IBM 3090/VF : An integrated environment for crash analysis," IBM Systems Journal 27, 541-560 (1988).
6. K.W. Neves, "The software Implications of Supercomputer Architecture," Proceedings of the 9th Conference on Electronic Computation, K.M. Will(Editor), ASCE, New York(1986).
7. H.S. Stone, High-Performance Computer Architecture, Addison-Wesley Publishing Company(1989).
8. C.E. Jeffcat and D.D. Wilmarth, "Attached scientific computers in structural analysis," Proceedings of the 9th Conference on Electronic Computation, K.M. Will(Editor), ASCE, New York(1986).
9. R.H. Perrott, Parallel Programming, Addison-Wesley Publishing Company(1987).
10. S. Lindsay, "Performance architecture Comparison of vector processors on large Scientific and commercial machines," CMG-87 Conference Proceedings(1987).
11. AS/XL Series Functional Characteristics, FE-95XL120-1, National Advanced Systems Corporation(1986).
12. D.J. Benson, "Vectorizing the right-hand side

assembly in an explicit finite element program," Comput. Meths. Appl. Mech. Engrg. 73, 147-152(1989).

13. K. So and V. Zecca, "Program locality of vectorized applications running on the IBM 3090 with Vector Facility," IBM Systems Journal 27, 436-452(1988).
14. J.O. Hallquist, Theoretical manual for DYNA3D, UCID-19401, Lawrence Livernore National Laboratory(1983).

(접수일자 : 1991. 1. 8)