PoLAPACK: 알고리즘적인 블록 기법을 이용한 병렬 인수분해 루틴 패키지

(PoLAPACK: Parallel Factorization Routines with Algorithmic Blocking)

최 재 영 †

(Jaeyoung Choi)

요 약 본 논문에서는 분산메모리를 가진 병렬 컴퓨터에서 밀집 행렬 연산을 위한 PoLAPACK 패키지를 소개한다. PoLAPACK은 새로운 연산 기법을 적용한 LU, QR, Cholesky 인수분해 알고리즘들을 포함하고 있다. 블록순환분산법으로 분산되어 있는 행렬에 알고리즘적인 블록 기법(algorithmic blocking)을 적용하여, 실제 행렬의 분산에 사용된 블록의 크기와 다른, 최대의 성능을 보일 수 있는 최적의 블록 크기로 연산을 수행할 수 있다. 이러한 연산 방식은 분산되어 있는 원래의 행렬 A의 순서를 따르지 않으며, 따라서 최적의 블록 크기로 연산을 수행한 후에 얻어진 해 x를 원래 행렬 분산법을 따라서 재배치하여야 한다. 본 연구는 Cray T3E 컴퓨터에서 구현하였으며 ScaLAPACK의 인수분해 루틴들과 그 성능을 비교·분석하였다.

Abstract LU, QR, and Cholesky factorizations are the most widely used methods for solving dense linear systems of equations, and for solving dense linear systems of equations, and have been extensively studied and implemented on vector and parallel computers. In this paper, we present PoLAPACK, which includes parallel LU, QR, and Cholesky factorization routines with an "algorithmic blocking" on 2-dimensional block cyclic data distribution. Too small or large a block size makes getting good performance on a machine nearly impossible. In such a case, getting a better performance may require a complete redistribution of the data matrix. With the algorithmic blocking, it is possible to obtain the near optimal performance irrespective of the physical block size. The routines are implemented on the SGI/Cray T3E and compared with the corresponding ScaLAPACK factorization routines.

1. 서 론

병렬 컴퓨터에서 대부분의 선형대수 알고리즘들은 알고리즘이 진행되면서 프로세서에 할당된 작업량(workload)이 불균등하게 되는 경우가 많다. 예를 들어 LU 인수분해 알고리즘은 데이타 행렬의 행과 열이 차례대로 없어지므로 결국 불균등하게 된다 [1, 2]. 데이타 행렬이 프로세서들에 분산되는 방식에 따라 병렬 알고리즘의 부하 균등과 통신 특성에 영향을 미치므로, 결국 알고리즘의 성능과 확장성(scalability)을 결정짓는다.

2차원적인 블록순환분산법(block cyclic data distribution)은 계산을 담당하는 각각의 프로세서가 $P \times Q$의 2차원적인 격자구조로 되어있다고 가정하고 그에 상응하게 데이타 행렬을 프로세서에 할당되도록 배열한다. 이 방식은 우수한 확장성과 부하 균등 특성을 가지고 있으므로 범용 선형대수 소프트웨어 라이브러리에서 사용되어 왔다 [3, 4].

병렬 컴퓨터들은 서로 아주 상이한 계산과 통신 비율을 가지고 있으므로, 알고리즘의 최대 성능을 낼 수 있는 최적의 블록의 크기는 서로 다르다. 따라서 알고리즘을 최대의 성능으로 실행시키려면, 데이타 행렬을 실행 전에 최적의 블록 크기로 미리 분산시켜 놓아야 한다.

$C \leftarrow C + A \cdot B$의 행렬 곱셈은 선형대수에서 가장

* 본 과제는 1999년도 한국과학기술 평가원의 핵심 소프트웨어 기술개발 사업(99-N-NS-05-A-02)의 지원을 받아 수행되었습니다.

† 종신회원 : 숭실대학교 컴퓨터학부 교수
choi@comp.ssu.ac.kr

논문접수 : 2000년 8월 8일
심사완료 : 2001년 3월 27일

기본적인 연산이다. 많은 병렬 곱셈 알고리즘이 2차원적인 블록 순환 분산법에서 제안되었다 [5, 6, 7]. 연속적인 *rank-K* 갱신을 수행하는 병렬 곱셈 알고리즘이 높은 성능과 확장성, 그리고 구현의 단순성을 보여주었다. 알고리즘에서 데이타 행렬은 블록의 크기를 K로 행렬 A는 열 블록으로, 행렬 B는 행 블록으로 나눈 후에 각각의 열과 행 블록들을 차례대로 곱하여 연산을 수행한다. 하지만 블록의 크기가 아주 작거나 혹은 아주 큰 경우에 계산과 통신을 효과적으로 중첩시킬 수 없기 때문에 좋은 성능을 기대하기가 어렵다.

DIMMA[7]는 최소공배수 (LCM) 개념을 DIMMA에 적용시켜서, 실제 행렬의 분산에 사용된 블록의 크기와 상관없이 최적의 블록크기로 연산을 수행할 수 있도록 하였다. DIMMA에서 만일 실제 분산에 사용된 블록의 크기가 최적의 블록 크기보다 작다면 이러한 작은 블록들을 큰 블록으로 결합시키고, 실제 블록의 크기가 크다면 커다란 블록들을 작은 블록으로 나누어 연산을 수행할 수 있도록 한다. 이것이 바로 알고리즘적인 블록의 기본 개념이다.

분산메모리를 가진 병렬 컴퓨터에서 알고리즘적인 블록 기법을 적용하여 인수분해 알고리즘들을 개발하려는 연구가 있었다. Lichtenstein과 Johnson[8]은 Connection machine CM-200에서 LU와 QR 인수분해를 위하여 열과 행 블록들을 블록순환적으로 제거하는 알고리즘을 개발하였다. 그러나 이 알고리즘은 블록 분산법으로 분산되어 있는 데이타에서만 사용할 수 있다.

미시시피주립대의 P. Bangalore[9]는 데이타 배분에 독립적인 알고리즘을 개발하려고 시도하였으나, 프로세서들에 실제로 분산된 행렬의 배열을 무시하고, 적합한 블록의 크기를 행렬의 순서에 따라 계산 패널을 재구성하여 계산을 진행하였다. 결과에 따르면, 데이타를 재분산시킬 때에 비교하여 성능이 더 나은 것으로 되어있지만, 프로세서들의 배열을 무시하고 계산 패널을 재구성하였으므로 트리 형식의 통신방법을 사용하였다. 결국 행렬의 재분산을 피하였지만 파이프라인 방식의 통신방식을 적용하지 않았으므로, 구조적으로 계산과 통신을 효과적으로 겹치게 할 수 없었다.

입력된 행렬과 컴퓨터의 특성 값들에 따라 컴퓨터가 수행할 때 실제 사용할 알고리즘을 선택하는 방법을 폴리알고리즘 (Polyalgorithm)이라고 한다 [10]. 우리는 컴퓨터가 블록의 크기를 탄력적으로 정하여 수행할 수 있는 PoLAPACK (Poly LAPACK)을 구현하여, 이를 이용하여 분산되어 있는 블록 크기 변화에 상관없이 항상 최대의 성능을 유지할 수 있도록 하였다.

PoLAPACK은 기본적으로 ScaLAPACK [11]의 구현 방식을 따랐지만, ScaLAPACK과는 다르다. ScaLAPACK은 전체 행렬의 크기를 나타내는 전역(global)값을 사용하였지만, PoLAPACK의 내부에서는 계산상의 복잡성으로 인하여 전역 값과 함께 각 프로세서들이 가지고 있는 지역(local)값을 함께 사용하여야 하였다.

본 연구는 대덕의 전자통신연구소에 있는 Cray T3E 컴퓨터에서 구현하였으며, ScaLAPACK의 LU, QR, Cholesky 인수분해루틴과 그 성능을 비교·분석하였다.

2. POLAPACK LU 인수분해 알고리즘

선형대수 연산을 위한 병렬 라이브러리의 연산루틴들은 사용하기 쉽고 이식성이 있으며, 그 성능이 충분히 좋아야 하고, 해결하고자 하는 문제의 크기와 프로세서들의 수가 커지면서 그에 비례하여 프로세서 당 성능이 일정(Scalable)하여야 한다. 이러한 목적을 달성하기 위해서는 연산을 수행할 때, 블록의 크기를 탄력적으로 변경시킬 필요가 있다. 본 논문에서는 기존의 ScaLAPACK의 LU, QR, Cholesky 인수분해 루틴을 기본으로 하여, 알고리즘적인 블록 기법을 이용한 PoLAPACK 인수분해 루틴들을 개발하였다.

ScaLAPACK 인수분해 루틴들에서는 기존의 방법들은 한 열의 프로세서들이 각각 자신에게 블록의 크기로 주어진 각각의 열방향 패널을 먼저 인수분해하고 이를 다음 프로세서들에게 전달시켜서 연산을 수행한다. 주어진 행렬의 블록의 크기가 아주 작거나 클 경우, 프로세서들이 자신의 성능을 제대로 발휘할 수 없으며, 이러한 경우 행렬을 완전히 재분산시켜야 하였다. 그러나 PoLAPACK에서는 행렬을 재분산시키지 않고도 효과적으로 연산을 수행할 수 있다.

먼저 PoLAPACK LU 인수분해 루틴을 중심으로 알고리즘적인 블록 기법을 알아보도록 한다.

기본적인 LU 인수분해 루틴은

$$A \cdot x = b \qquad (1)$$

의 연산식에서 A를 LU 인수분해한 후에, (1)식의 해 x를 찾는 것이다. 즉 A를 $P \cdot A = L \cdot U$로 인수분해하고, $L\ U \cdot x = P \cdot b = b_1$의 식으로 변형하여,

$$L \cdot y = b_1 \qquad (2)$$

을 이용하여 y를 구한 후에

$$U \cdot x = y \qquad (3)$$

로부터 x를 구하게 된다. 기본적으로 LAPACK과 ScaLAPACK, 그리고 거의 모든 다른 구현들은 주어진 선형연산식을 $P \cdot A = L \cdot U$로 변형한 후에, 이를 이용

하여 x 값을 얻게된다.

ScaLAPACK의 LU 인수분해 루틴을 Cray T3E에서 블록의 크기를 변경시키면서 성능의 변화를 관찰하였다. 8×8의 프로세서에서 N = *1,000*에서 *16,000*까지 블록 크기를 Nb = *1, 6, 24, 36, 60*으로 변경시켰을 때 성능의 변화는 그림 1과 같다. 그림 1에서 N_b = *36*일 때 최대의 성능을 보였고, N_b = *60*, 혹은 *24*일 때는 2-4% 정도의 성능 저하를 보였다. N_b = *6*일 때는 40%의 성능이 저하되었고, N_b = *1*일 때는 80% 이상의 성능이 저하되었다. 그림에서 보듯이 만일 N_b = *1*로 데이타가 분산되어있다면, 효과적인 연산을 위해 데이타를 완전히 재분산시켜야 할 것이고, 기존의 데이타 행렬을 저장시키기 위한 공간도 필요하게 될 것이다.

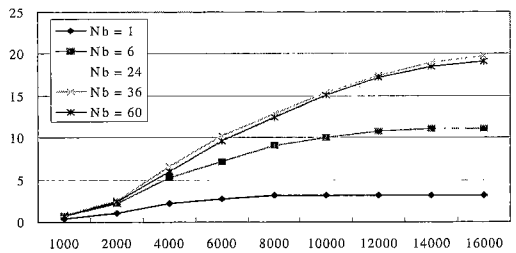


그림 1 8×8의 Cray T3E에서 블록 크기 변화에 따른 ScaLAPACK의 LU 인수분해 루틴의 성능 변화

그림 1에서 보듯이 ScaLAPACK에서는 실제 분산에 사용된 블록의 크기 변화에 따라 성능이 달라진다. 그러나 PoLAPACK에서는 블록 크기 변화에 상관없이 항상 N_b=*36*의 최적의 성능을 유지하기 위하여 알고리즘적인 블록 기법(Algorithmic Blocking)을 적용하였다.

행렬 A가 2차원적으로 프로세서들 위에 블록순환분산법으로 분산되어 있으므로, 이것은 프로세서들의 조합에 따라서 A의 블록들을 행과 열 방향으로 새로운 순열을 이용하여 재분산시킨 것이다. 행방향과 열방향은 각각 왼쪽과 오른쪽에서 순열행렬을 곱한 것과 같다. 즉 A의 양변 각각에 순열행렬을 곱하여 $P_P \cdot A \cdot P_Q{}^T$로 변형시킨 후 LU 인수분해시킬 수 있다면, 행렬 A를 고정된 블록들의 순서에 의하지 않고, 새로운 순서로 연산을 수행할 수 있다. 다시 말해서 A의 블록의 크기가 아주 작아 프로세서들의 성능을 충분히 발휘할 수 없을 때, 처음에 정해진 행렬의 순서에 상관없이 한 열의 프로세서가 가지고 있는 여러 개의 블록들을 합하여 하나의 커다란 블록으로 조합하여 연산할 수 있고, 또한 A의 블록의 크기가 아주 클 때에도, 커다란 블록을 쪼개어 작은 블록으로 만들어 연산을 수행하므로써, 항상 최대의 성능을 보일 수 있다.

이것을 식으로 표현하면, $A \cdot x = b$는 다음과 같이 변형된다.

$$(P_P\ A\ P_Q{}^T) \cdot (P_Q\ x) = P_P \cdot b \qquad (4)$$

여기서 P_P와 P_Q는 순열행렬이며, $P_X{}^T \cdot P_X = P_X{}^{-1} \cdot P_X = I$가 된다. (여기서 X는 P이거나 Q이다.) $P_P \cdot A \cdot P_Q{}^T = A_1$으로, $P_Q\ x = x_1$으로 놓고 A_1을 $P_1\ A_1 = P_1 \cdot (P_P A P_Q{}^T) = L_1 \cdot U_1$로 LU 인수분해시킨 후에 x를 구하도록 한다. 그러면, 위의 (4)식은

$$L_1 \cdot U_1\ (P_Q\ x) = L_1 \cdot U_1 \cdot x_1 = P_1 \cdot (P_P\ b) = b_1 \qquad (5)$$

으로 변형된다. (2)와 마찬가지로

$$L_1 \cdot y_1 = b_1 \qquad (6)$$

을 이용하여 y_1을 구한 후에

$$U_1 \cdot x_1 = y_1 \qquad (7)$$

으로 x_1을 구하고, 마지막으로

$$P_Q \cdot x = x_1 \qquad (8)$$

이므로, x를 구할 수 있다.

$P_1 \cdot A_1 = P_1 \cdot (P_P A P_Q{}^T) = L_1 \cdot U_1$으로 연산하는 것은 위에서 설명한대로 블록들을 합치거나 쪼개어서 실제로 행렬을 분산시킬 때 사용된 블록의 크기에 무관하게 최적의 블록 크기를 만들어 연산을 수행한다. 원하는 해 x를 구하기 위해서 (5)식에 보인 것처럼 $P_1 \cdot (P_P\ b) = b_1$으로 변형시킨 후에 (6), (7)식을 순서대로 적용시켜야 하며, 그 해 x는 (8)식에 보인 것처럼, x_1에 P_Q를 이용하여 다시 재분산시키므로써 구할 수 있다. 그러나 x도 P_P에 의하여 이미 분산되어 있으므로 x보다는 $P_P \cdot x$를 구해야 한다.

$$P_P \cdot x = P_P \cdot P_Q{}^T \cdot x \qquad (9)$$

로부터 해 $P_P \cdot x$를 구할 수 있다.

3. PoLAPACK LU 인수분해 루틴의 구현

PoLAPACK LU 인수분해 루틴의 구현은 다음의 세 부분으로 구성되어 있다.

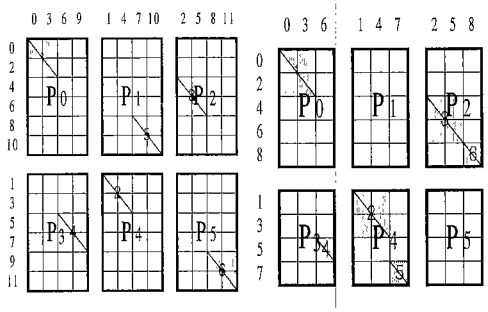
[1] LU 인수분해 부분: Right-Looking version의 LU를 구현한다.

[2] 삼각행렬 해 루틴 부분: Li와 Coleman의 알고리즘[12]을 2차원적으로 구현하여야 한다. LAPACK에서는 BLAS의 DTRSM이 이를 담당하지만, 각 대각선상에 위치한 블록들이 규칙성있게 위치하지 않을 수 있으므로 구현이 매우 복잡하고 까다롭다.

[3] 해 재분배 부분: 해 x를 재분배하는 부분은 (9)에서 보인 것과 같이 $P_P \cdot P_Q{}^T \cdot x$로 재분배시켜야 한다. 그렇지만 행렬 A가 블록순환분산법으로 분산되어

있고 $P = Q$라면 $P_P = P_Q$가 되어 $P_P \cdot P_Q^T \cdot x = x$이므로, 이 부분은 필요하지 않게 된다. 그렇지만 $P \neq Q$라면 이 문제는 블록벡터의 전치를 두 번 수행하는 all-to-all personalized communication 문제로 바뀐다.

[1]에서 행렬 A는 블록순환분산법으로 분산되어 있지만, 연산은 원래 행렬의 순서를 따르지 않는다. 그리고 계산이 진행되면서도 각 프로세서가 가지고있는, 남아있는 행렬의 크기를 정수 연산으로 쉽게 찾을 수 없다. 그러므로 연산이 시작되기 전에 각 프로세서들은 다른 프로세서들이 가지고 있는 행렬의 크기를 계산하고, 이를 연산이 진행되는 동안 계속 갱신하여, 다른 프로세서에 남아있는 데이타의 크기를 항상 알 수 있도록 구현하였다.



(a) 12×12 블록이 2×3 프로세서에 있을때 (b) 9×9 블록이 2×3 프로세서에 있을경우

그림 2 블록크기에 따르는 계산 진행 과정

그림 2는 PoLAPACK LU의 계산 진행 과정을 보여준다. 그림 2에서는 2개의 블록을 한번에 연산한다고 가정하였다 (예를들면 N_b=4, N_b'=8일 경우). 그림 2 (a)는 *12×12* 블럭이 *2×3* 프로세서에 분산되어 있는 경우로 블록순환분산법의 계산과정을 따르기 때문에 각 계산과정이 한 블록씩 진행되면서 대각 행렬 블록의 위치가 한 행과 한 열씩 규칙적으로 증가하였다. 그러나 그림 2 (b)는 *9×9* 블록이 *2×3*프로세서 위에 분산되어 있는 경우로 P_3이 (A(5,6))의 대각 블럭을 계산한 후에 그 다음 대각 블럭이 P_1에 위치하지 않고 P_4로 (A(7,7)), 그 다음에 P_2로 (A(8,8))로 옮겨왔다. 따라서 PoLAPACK의 계산과정은 ScaLAPACK보다도 훨씬 복잡하며 구현하기가 용이하지 않다.

그리고 PoLAPACK에서는 각각의 프로세서가 가지고 있는 데이타의 크기는 실제 분산에 사용된 블록의 크기에 의해 결정되며, 연산이 진행되면서 연산이 종료되지 않았는데도 불구하고 행과 열의 프로세서들이 데이타를 가지고 있지 않을 경우가 있다. 그림 3에서 보듯이 '40 CONTINUE'의 아래 부분이 연산할 데이타가 남아있는 프로세서들을 찾아 연산을 계속하여 수행할 수 있도록 한다. 기본적인 루틴보다 다소 복잡해 보이지만, $P \neq Q$인 경우에도 연산을 성공적으로 수행하도록 구현되어 있다.

```
      DO 20 J = 1, N, NB
         IB = MIN( M-J+1, NB )
         JB = MIN( N-J+1, IB )
         NXTROW = MOD( ICURROW+1, NPROW )
         NXTCOL = MOD( ICURCOL+1, NPCOL )
          ......
         ICURROW = NXTROW
         ICURCOL = NXTCOL
   20 CONTINUE
```

(a) 기본적인 Loop 방식

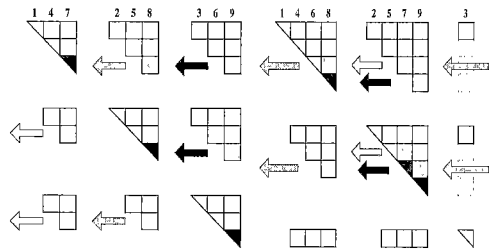
```
      NDONE = 0
      IB = MIN( IWORK(IPP+ICURROW), NNB )
      JB = MIN( IWORK(IPQ+ICURCOL), NNB )
*
   20 CONTINUE
      LB = MIN( IB, JB )
      NXTROW = MOD( ICURROW+1, NPROW )
      NXTCOL = MOD( ICURCOL+1, NPCOL )
      IF( LB.EQ.0 ) GO TO 40
         ......

      IWORK(IPP+ICURROW) = IWORK(IPP+ICURROW) - LB
      IWORK(IPQ+ICURCOL) = IWORK(IPQ+ICURCOL) - LB
*
   40 CONTINUE
      NDONE = NDONE + LB
*
      IF( NDONE.LT.MINMN ) THEN
         KB = IB - JB
         IF( KB.GT.0 ) THEN
            IB = IB - LB
         ELSE
            ICURROW = NXTROW
   50       CONTINUE
            IB = IWORK(IPP+ICURROW)
            IF( IB.LE.0 ) THEN
               ICURROW = MOD( ICURROW+1, NPROW )
               GO TO 50
            ELSE
               IB = MIN( IB, NNB )
            END IF
         END IF
*
         IF( -KB.GT.0 ) THEN
            JB = JB - LB
         ELSE
            ICURCOL = NXTCOL
   60       CONTINUE
            JB = IWORK(IPQ+ICURCOL)
            IF( JB.LE.0 ) THEN
               ICURCOL = MOD( ICURCOL+1, NPCOL )
               GO TO 60
            ELSE
               JB = MIN( JB, NNB )
            END IF
         END IF
*
         GO TO 20
      END IF
```

(b) PoLAPACK의 Loop 방식

그림 3 PoLAPACK의 Loop 방식 비교

[2]에서의 PoLAPACK에서 삼각행렬의 해를 구하는 Solver 루틴(PoDTRSM)의 구현은 매우 복잡하고 까다롭다. 기본적으로 ScaLAPACK의 해를 구하는 루틴은 그림 4에서처럼 Li와 Coleman의 알고리즘[12]을 2차원적으로 구현한 것이다. 기본적인 Li와 Coleman의 알고리즘은 삼각 행렬 T가 일렬로 나열된 Q개의 프로세서에서 삼각행렬의 해를 구하려고 할 때, 해를 구한 프로세서가 $Q-1$개의 데이타를 먼저 계산한 후 프로세서들에게 전달한 후, 나중에 나머지 데이타들을 계산하는 것이다. ScaLAPACK에서는 데이타가 2차원적으로 분산되어 있으므로 해를 구하는 열의 프로세서들이 $Q-1$개의 블록들을 먼저 계산한 후 전달해주고, 후에 나머지 블록들을 각자 계산한다. 한편 PoLAPACK에서는 그림 4에서 보듯이 해를 구하는 방식이 ScaLAPACK의 그것과 비교하여 훨씬 복잡하다.

그림 4 (a)는 3x3의 9개 프로세서에 분산되어 있는 상삼각 행렬을 해결하는 과정을 보여주고 있다. 그림의 ScaLAPACK 삼각행렬 Solver에서는 실제 분산되어 있는 블록의 크기가 1이고 연산하는 블록의 크기가 역시 1로 블록들을 차례로 돌아가면서 연산하도록 한다. 그러나 PoLAPACK에서는 그림 4 (b)에서 보는 것처럼 실제 분산되어 있는 블록의 크기가 4이고 연산하는 블록의 크기가 역시 1일 수 있고, 이러한 경우에도 문제를 해결할 수 있도록 알고리즘을 구현하여야 한다. 따라서 PoLAPACK에서는 해를 계산하는 열 프로세서가 선행하는 $Q-1$ 블록들을 미리 계산하여, 각각의 프로세서가 얼마나 먼저 계산하고 보낼지를 결정한다.



(a) ScaLAPACK 삼각행렬 Solver (b) PoLAPACK 삼각행렬 Solver

그림 4 ScaLAPACK과 PoLAPACK의 삼각행렬Solver

한편 [3]의 PoLAPACK 해 재분배 루틴은 (9)식에서와 같이 $P_p \cdot x = P_p \cdot P_q^T \cdot x$로 구하였던 해 x를 두 번 전치시켜야 한다. 이를 구체적으로 살펴보면, 아래의 그림 5와 같다.

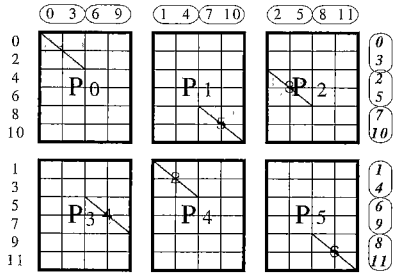


그림 5 PoLAPACK의 해 재분배 루틴의 통신 과정

그림 5는 실제 분산되어 있는 블록의 크기는 $N_b = 1$이고 연산하는 블록의 크기는 $N_{opt} = 2$로 계산된 결과를 보여주고 있다. 계산 결과 분산된 해 x는 그림의 오른쪽에 있는 블록의 배열이 되지만 ($P_P\ P_Q^T\ x$), 이를 재분배하여 왼쪽의 원래 형태($P_P\ x$)로 만들어 주어야 한다. 그림 5에서 보듯이 오른쪽에 있는 벡터 x를 위에 있는 벡터로 전치시킨 후에 그것을 다시 왼쪽에 있는 벡터로 전치시킨다면, 원하는 배열을 가진 해를 구할 수 있다. 그러나 오른쪽에 있는 해를 왼쪽에 있는 해로 옮기는 것은, 전체 프로세스들이 연루되지 않고 해를 가지고 있는 한 열의 프로세스들 안에서만 일어날 수 있다. 벡터의 전치를 한 열의 프로세스에서 수행하는 것은, 각 프로세스가 다른 모든 프로세스들에게 각기 다른 데이타를 전달하는 all-to-all personalized communication 문제이다.

그 계산 과정을 보여주는 가상코드는 그림 6에서 보여주고 있다. 한 열의 프로세서에서 이 문제를 해결하기 위하여 먼저 각 프로세서들은 자신이 가지고 있는 데이타들의 순서를 계산한다. 그리고는 인덱스만을 두 번 전치시켜서 그 데이타가 보내질 프로세서와 프로세서내에서의 위치를 계산한다. 그 인덱스를 따라 프로세서는 보내질 프로세스내에서의 위치 정보와 데이타를 보내고, 받는 프로세스들은 받은 즉시 데이타를 위치정보를 따

```
Computing Procedure for Pi
1. Compute the local & global indices of data
   in each processor
2. 1st transpose its indices of data (with Nob)
      (not transpose the real data)
3. 2nd transpose its indices of data (with Nb)
      (not transpose the real data)
4. For i=0 to p-1
   i) Copy & send data & indices to P(i+1)
   ii) Receive & move data & indices from P(i-1)
```

그림 6 해 재분배 알고리즘의 가상 코드

라 저장시킨다.

그림 6에서 1~3의 계산 과정을 보다 단순화시키기 위하여 최소 공배수(LCM)의 개념을 이용한다. $P \times Q$의 프로세스에서 실제 데이타는 N_b로 분산되어 있고 N_{opt}로 연산하였다면, 이들의 반복을 이루는 최소한의 수는 $LCMNUM = LCM(P, Q) \cdot LCM(N_b, N_{opt})$이다. 따라서 인덱스의 연산은 이들의 반복이며 반복된 인덱스의 연산은 생략할 수 있다. 그러나 그림 6에서 보듯이, 오른쪽의 인수분해 연산 후에 각 프로세스가 가지고 있는 데이타의 개수는 이미 결정되어 있다 (반드시 왼쪽과 같아야 한다.) 즉 최소공배수의 개념을 적용하여 인덱스를 $LCMNUM$ 만큼만 계산하고 나머지들은 1~3의 계산과정으로 따로 다시 계산하여야 한다.

PoLAPACK LU 인수분해 루틴을 8×8 Cray T3E에서 성능을 측정하였다. 그림 7에서 확인할 수 있듯이 PoLAPACK LU 인수분해 루틴은 실제 분산된 블록 크기 N_b와는 상관없이 블록크기 $N_{opt}=8$에서의 항상 최적의 성능을 보이고 있다. 각 프로세서들이 N_b에 상관없이 항상 같은 크기의 데이타들을 가지고 있지 않기 때문에 그림 7에서 약간의 무시해도 좋을 정도의 차이가 보인다.

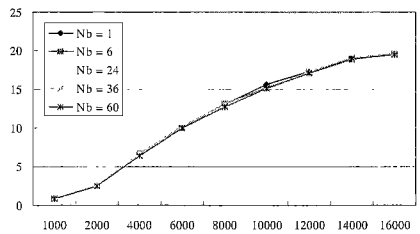


그림 7 8×8 Cray T3E에서의 PoLAPACK LU의 성능 ($N_{opt}=36$)

4. PoLAPACK QR 인수분해 루틴

기본적으로 QR 인수분해 루틴은

$$min \; || A\,x - b ||_2 \qquad (10)$$

를 만족시키는 x를 찾는 것이다. 이를 위하여 A를 $Q \cdot R$로 인수분해하도록 한다. 여기서 Q는 직교 (othogonal) 행렬이고 R은 상삼각 (Upper Triangular) 행렬이다. PoLAPACK LU 인수분해 알고리즘과 마찬가지로 A는 $P \times Q$의 프로세서들에 분산되어 있으므로, A를 $Q \cdot R$로 인수분해하는 대신 $P_P \cdot A \cdot P_Q{}^T$를 $Q_1 \cdot R_1$으로 인수분해하도록 한다.

$$P_P\, A\, P_Q{}^T \Rightarrow Q_1 \cdot R_1 \qquad (11)$$

그러면 (10)식은 다음과 같이 변형시킬 수 있다.

$$(P_P\, A\, P_Q{}^T) \cdot (P_Q\, x) = P_P\, b = b_0 \qquad (12)$$

$$R_1\, x_1 = Q_1{}^T\, b_0 = b_1 \qquad (13)$$

(4)식으로부터 x_1을 구하고 $P_Q\, x = x_1$이다. 우리가 구하고자하는 x는 LU 인수분해와 마찬가지로 P에 의해 분산되어 있으므로 $P_P\, x$를 찾아야 한다. 결국

$$P_P \cdot x = P_P \cdot P_Q{}^T\, x_1 \qquad (14)$$

로부터 $P_P \cdot x$를 구하도록 한다.

PoLAPACK QR 인수분해도 LU 인수분해와 같은 방식으로 구현하였다. 그림 8은 ScaLAPACK과 PoLAPACK QR 인수분해 루틴을 Cray T3E 컴퓨터를 사용하여 그 성능을 비교한 것이다.

그림 8에서 보듯이 ScaLAPACK QR 인수분해 알고리즘은 블록의 크기에 따라 성능이 차이가 있었지만, PoLAPACK QR 루틴은 실제로 분산되어있는 행렬의 크기에 상관없이 항상 일정한 성능을 유지할 수 있음을 보여주고 있다. PoLAPACK QR 인수분해 알고리즘에서는 알고리즘적인 블록의 크기를 24로 고정하여 두고 각 프로세서가 가지고 있는 행렬을 연산하도록 하였기 때문이다.

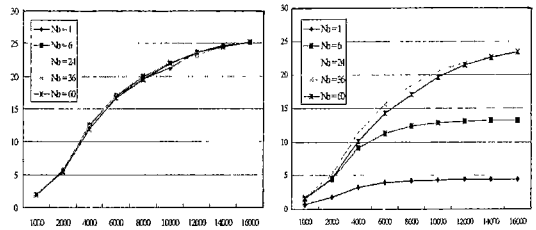


그림 8 8x8의 Cray T3E에서 블록 크기 변화에 따르는 QR 인수분해 루틴의 성능($N_{opt}=24$)

5. PoLAPACK Cholesky 인수분해 루틴

Cholesky 인수분해 루틴은 symmetric positive definite (약어로 SPD로 표시)인 행렬 A를 하나의 하삼각 행렬과 그것의 전치행렬의 곱으로 인수분해하는 것을 말한다. 즉 $A = L \cdot L^T$로 (혹은 만일 U가 상삼각 행렬이라면 $A = U^T \cdot U$로) 나타낼 수 있다. SPD인 A의 선형방정식 $A \cdot x = b$에서 $A = L \cdot L^T$로 인수분해된다면,

$$L \cdot y = b \qquad (15)$$

을 이용하여 y를 구한 후에

$$L^T \cdot x = y \qquad (16)$$

로부터 x를 구하게 된다.

PoLAPACK Cholesky 인수분해에서는 A가 대칭행렬이므로 $P \neq Q$인 경우에 LU와 QR 인수분해에서처럼 $P_P\, A\, P_Q{}^T$가 대칭 행렬을 유지할 수 없으므로, 알고

리즘적으로 블록의 크기를 변경하여 사용할 수 없다. 즉 $P = Q$인 경우로 한정하여

$$P_P\ A\ P_P{}^T \Rightarrow L_1 \cdot L_1{}^T \quad (17)$$

그러면

$$(P_P\ A\ P_P{}^T) \cdot (P_P\ x) = P_P\ b = b_0 \quad (18)$$

$$L_1\ y_1 = b_1 \quad (19)$$

$$L_1{}^T\ x_1 = y_1 \quad (20)$$

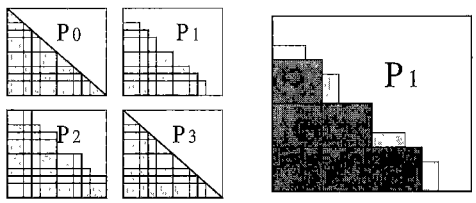
(19)식으로부터 y_1을 구하고, 다시 (20)식을 이용하여 x_1을 구한다. 구하려는 x는 역시 P에 의해 분산되어 있으므로 $P_P\ x$를 찾아야 한다. 결국

$$P_P \cdot x = P_P \cdot P_P{}^T\ x_1 = x_1 \quad (21)$$

즉, PoLAPACK Cholesky 인수분해에서는 P=Q이므로 해를 재분배하는 과정이 생략된다.

$P = Q$인 경우에 $P_P\ A\ P_P{}^T$도 대칭 행렬이지만, 흔히 대칭 행렬의 경우 하삼각 혹은 상삼각 부분에만 행렬이 존재하며, 그 나머지 부분을 갱신하여서는 안된다. 따라서 알고리즘적인 블록을 이용하기 위하여는 LU나 QR 인수분해에서 적용하던 방식과는 다른 기교가 요구된다. PoLAPACK Cholesky 인수분해를 하는 행렬 $P_P\ A\ P_P{}^T$에서 연산하는 블록의 크기를 변경시킬 경우에 새로운 블록의 삼각 행렬은 원하는 데이타가 없으므로 계산 전에 내부적으로 이를 채워주어야 한다.

그림 9는 2×2 프로세서에서의 실제 행렬 분산에 사용된 블록의 크기 N_b=2이고, 연산하는 블록의 크기는 N_{opt}=3일 경우에, 하삼각 행렬이 어떻게 변형되어야 하는지를 그림으로 보여준다. 실제 행렬 분산에 사용된 블



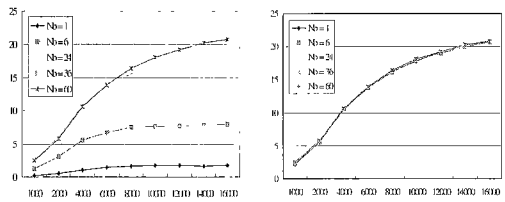
(a) 2×2 프로세서에서 데이터 분산 (b) P_1에서의 데이타 이동

그림 9 블록 연산 단위 변경에 따른 하삼각 행렬에서의 데이타 보존

록의 크기가 2라면 대칭 행렬은 각 프로세스에 연한 바탕색을 가진 블록에만 저장되어 있다. 만일 연산하는 블록의 크기가 3이라면 하삼각 행렬의 데이타는 진한 실선으로 표시된 블록안에 위치하여야 한다.

따라서 그림 9 (b)의 P_1의 예에서 보듯이 남아있는 원래 블록의 행렬은 P_2로 보내져야 한다. 물론 동시에 P_1은 P_2로부터 자신의 필요한 데이타를 받아야 한다. 앞에서 설명한대로 PoLAPACK Cholesky 인수분해 알고리즘은 $P = Q$인 경우에서만 사용할 수 있으므로 대각선에 위치하고 있지 않은 프로세스들은 대각선과 대칭되는 프로세스들과 데이타를 교환하여야 한다. 계산을 위한 최적의 블록의 크기로 연산을 수행한 후에 데이타를 교환하였던 프로세스들은 원래의 행렬을 복원하기 위하여 자신의 상대와 다시 데이타를 교환하여야 한다.

그림 10은 8×8의 Cray T3E에서 ScaLAPACK과 PoLAPACK Cholesky 인수분해 알고리즘의 성능을 보여주고 있다. 예상하였던대로 PoLAPACK 루틴은 실제로 분산되어있는 행렬의 크기에 상관없이 최적의 성능을 보여주고 있다. 그림 (b)는 N_{opt}=60으로 PoLAPACK Cholesky 인수분해 루틴을 수행한 결과이다.



(a) ScaLAPACK Cholesky 인수분해 루틴의 성능 (b) PoLAPACK Cholesky 인수분해 루틴의 성능

그림 10 8×8의 Cray T3E에서 블록 크기 변화에 따른 Cholesky 인수분해 루틴의 성능(N_{opt}=36)

6. 결 론

분산 메모리를 가진 병렬 컴퓨터들은 프로세서의 성능에 대비한 데이타 통신의 성능의 비율의 차이가 크게 다르기 때문에 알고리즘에 따라 각각의 컴퓨터에 알맞은 최적의 블록의 크기를 맞추어 주어야 한다. 경우에 따라서는 블록의 크기가 너무 작은 경우 혹은 너무 큰 경우에 성능이 크게 떨어져서, 행렬을 완전히 재분산시켜야 하였다. 그러나 본 논문에서는 이러한 문제를 해결하고 분산된 데이타의 크기에 상관없이 항상 최고의 성능을 얻기 위해서 알고리즘적인 블록 기법을 사용하여 항상 최대의 성능을 얻을 수 있도록 하였다.

일반적으로 ScaLAPACK과 같은 루틴들에서는 각 프로세서들이 가지고 있는 행렬 A의 열을 따라 인수분해를 하였으므로, A의 블록의 크기에 따라 블록 연산을 수행하여야 한다. 그러나 PoLAPACK 인수분해 루틴들에서는 데이타 행렬 A의 양쪽에 순열행렬 P_P와 $P_Q{}^T$를 곱하여 완전히 새로운 행렬 $P_PAP_Q{}^T$를 만든 후에 인수분해하였다. 즉 PoLAPACK LU에서는 $P \cdot A = L \cdot U$

대신에, $P_1 \cdot A_1 = P_1 \cdot (P_P A P_Q) = L_1 \cdot U_1$으로 연산하여, 프로세서들의 배열에 상관없이 아주 효과적인 연산을 수행할 수 있도록 하였다. 원래 주어진 연산의 순서를 바꾸어 연산을 수행할 수 있으므로, 행렬 A의 실제 배열에 상관없이 마음대로 블록들을 합치거나 쪼갤 수 있고, 따라서 항상 최적의 블록 크기로 연산을 수행할 수 있다. 해 x는 $P_1\ (P_P\ b) = b_1$으로 변형시킨 후에 L_1, U_1에 관하여 차례로 삼각 행렬의 해를 구한 후, P_P와 P_Q를 이용하여 다시 재분산시킴으로써 구할 수 있다. 여기서 중요한 점은 $P_P\ A\ P_Q{}^T$를 만들기 위하여 실제적인 데이타의 전송은 없으며, 각 프로세서가 가진 데이타의 크기에 따라 최적의 블록 크기로 연산을 수행할 수 있다는 점이다.

Cray T3E에서 ScaLAPACK과 PoLAPACK의 LU, QR, Cholesky 인수분해 루틴들의 성능을 비교하여 보면 블록 크기 N_b에 따라 성능차이를 보이는 ScaLAPACK과는 달리, PoLAPACK 루틴들은 블록 크기와는 무관하게 항상 최대의 성능을 보였다.

참 고 문 헌

[1] Jaeyoung Choi, Jack J. Dongarra, Susan Ostrouchov, David W. Walker, and R. Clint Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," Scientific Programming, Vol.5, pp.173-184, 1996.

[2] J. Dongarra and S. Ostrouchov, "LAPACK Block Factorization Algorithms on the Intel iPSC/860," LAPACK WOrking Note 24, Technical Report CS-90-115, University of Tennessee, 1990.

[3] J. Choi, J. Dongarra, and D. Walker, "The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers," Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France), pp.3-15. Elsevier Science Press Publishers, Sept. 7-8, 1992.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to Parallel Computing," The Benjamin/Cummings Publishing Co. Redwood City, CA. 1994.

[5] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers," Concurrency: Practice and Experience, Vol.6, No.7, pp.543-570, October, 1994.

[6] R. van de Geihn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," LAPACK WOrking Note 99, Techinical Report CS 95-286, University of Tennessee, 1995.

[7] Jaeyoung Choi, "A New Parallel Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers," Concurrency: Practice and Experience, Vol.10, No.8, pp.655-670, August, 1998.

[8] W. Lichtenstein and S. L. Johnsson, "Block-Cyclic Dense Linear Algebra," SIAM J. of Sci. Stat. Computing, 14 (6), pp.1259- 1288, 1993.

[9] P. V. Bangalore, The Data Distribution- Independent Approach to Scalable Parallel Libraries, Master Thesis, Mississippi State University, 1995.

[10] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK, A Portable Linear Algebra Library for Distributed Memory Computers-Design Issues and Performance," Proceedings of the Supercomputer 96, November 1996, IEEE Computer Society Press, 1996.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, "ScaLAPACK Users' Guide," SIAM, Philadelphia, 1997.

[12] G. Li and T. F. Coleman, "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor," SIAM J. of Sci. Stat. Computing, Vol. 9, pp.485-502, 1986.

최 재 영

1984년 서울대학교 제어계측공학과 학사. 1986년 미국 남가주대학교 전기공학과 석사(컴퓨터공학). 1991년 미국 코넬대학교 전기공학부 박사(컴퓨터공학). 1992년 1월 ~ 1994년 2월 미국 국립 오크리지 연구소 연구원. 1994년 3월 ~ 1995년 2월 미국 테네시 주립대학교 연구교수. 1995년 3월 ~ 현재 숭실대학교 정보과학대학 컴퓨터학부 부교수. 관심분야는 병렬/분산처리, 병렬알고리즘, 시스템 소프트웨어