클러스터 컴퓨팅 환경에서 병렬루프 처리를 위한 재구성 가능한 부하 및 성능 균형 방법
(A Reconfigurable Load and Performance Balancing Scheme for Parallel Loops in a Clustered Computing Environment)

김 태 형 †
(Tae-Hyung Kim)

요 약 부하 불균형은 병렬처리에 있어서 좋은 성능을 얻기 위한 주요한 방해 요소 중의 하나이다. 전역(全域) 부하균형 기법은 하나의 응용에서 발생된 병렬 태스크를 취급하는데 적절하지 않다. 동적 루프 스케줄링 기법은 공유 메모리 멀티프로세서 병렬구조에서 병렬 루프의 부하균형에 효과적인 것으로 알려져있다. 하지만 이 기법의 중앙집중적 특성은 워크스테이션 클러스터 환경에서 프로세서 수가 상대적으로 많지 않은 경우에도 병목현상을 일으킬 수 있는 요인이 된다. 워크스테이션 클러스터 환경에서의 통신 오버헤드는 공유 메모리 멀티프로세서 병렬 구조와 비교할 때 수십배의 차이가 생기기 때문이다. 더구나 병렬 루프에서 발생하는 단위 태스크가 불규칙적인 작업량을 갖는 경우에는 기본 루프 스케줄링 기법의 단점을 보완한 개선된 방법들을 적용할 수가 없다. 본 논문에서는 이러한 불규칙적인 작업량을 갖는 병렬 루프를 서로 다른 성능을 갖는 워크스테이션들의 네트워크 환경에서 효율적으로 부하를 분배하기 위한 재구성 가능한 분산 부하 균형 기법을 제시한다. 이러한 재구성 가능한 기법은 전통적인 부하균형 방법과 함께 성능균형을 가능하게 함으로써 전체수행시간을 최소화할 수 있음을 보였다.
키워드 : 클러스터 컴퓨팅, 병렬루프, 부하균형, 성능

Abstract Load imbalance is a serious impediment to achieving good performance in parallel processing. Global load balancing schemes cannot adequately manage to balance parallel tasks generated from a single application. Dynamic loop scheduling methods are known to be useful in balancing parallel loops on shared-memory multiprocessor machines. However, their centralized nature causes a bottleneck for the relatively small number of processors in a network of workstations because of order-of-magniture differences in communication overheads. Moreover, improvements of basis loops scheduling methods have not effectively dealt with irregularly distributed workloads in parallel loops, which commonly occur in applications for a network of workstations. In this paper, we present a new reconfigurable and decentralized balancing method for parallel loops on a network of workstations. Since our method supplements performance balancing with those tranditional load balancing methods, it minimizes the overall execution time.
Key words : cluster computing, parallel loops, load balancing, performance

1. 서 론

일반적인 분산 병렬 프로그램에서 태스크들을 다중의 분산 프로세서에서 동시에 처리하기 위하여 어떻게 분산할 수 있는가가 중요하다. 특히 어떤 프로세서들이 할 일이 없어서 쉬고 있다면 이 자원은 작업 수행을 더 빠르게 하는데 사용될 수 있기 때문에 이러한 부하 불균형이 높은 성능을 성취하는데 심각한 방해 요소가 된다는 것은 자명하게 이해된다. 물론 전체 운영체제 측면에서 보는 전체적인 부하균형이 매우 중요한 주제이기는 하지만, 본 논문에서는 하나의 응용 프로그램에서 발생되는 병렬 태스크들을 여러 프로세서에 어떻게 균등하게 할당해 줄 것인가 하는 문제에 초점을 맞추기로 한다. 이 경우에는 특히 하나의 프로그램이 최종적으로 완료되는 시간을 최소화하는 것이 각 병렬 태스크들이 완료되기 위해 필요한 평균 반응 시간(average response time)보다 더 중요하기 때문에 각 프로세서들이 처리할

† 종신회원 : 한양대학교 전자컴퓨터공학부 교수
tkim@cse.hanyang.ac.kr
논문접수 : 2003년 6월 9일
심사완료 : 2003년 11월 13일

부하량 측면에서 균형을 이루도록 노력하는 것보다는, 각 프로세서가 쉬지 않고 계산을 계속하도록 유지시키는 것이 더 중요하다는 것을 인식해야 한다. 즉, 하나의 응용 프로그램에서 발생하는 병렬 태스크들을 공평하게 나누어도 각 프로세서에서 작동되는 다른 프로세스들을 고려하면 하나의 워크스테이션 클러스터에 속해 있는 각 워크스테이션들이 동일한 시간에 작업을 완료할 수가 없기 때문이다. 일반적으로 처리할 부하를 공평하게 유지하는 것이 전체 처리 시간을 단축하는데 도움을 주기는 하지만 부하균형을 위해 과도한 스케줄링 오버헤드가 발생할 수 있으므로 이것이 항상 최종 수행 시간을 감축시킬 수 있는 것은 아니다[1].

프로그램의 입장에서 보면, 응용 프로그램의 병렬성의 최대 근원에 해당하는 부분이 루프(loops)이다. 이러한 루프를 **병렬 루프(parallel loops) 또는 DOALL-loop** 라고 부른다. 병렬 루프에는 각 루프내의 반복주기(iterations)들이 서로 독립적으로 존재해야 한다. 병렬 루프에서 발생하는 병렬 태스크들을 특정 병렬 프로세서 구조에 할당하여 전체 수행 시간을 최소화하는 방법은 루프 스케줄링 문제(loop scheduling problem)로 알려져 있다[2-6]. 만일 반복주기들간에 데이타 의존성이 있다면 병렬수행이 불가능하기 때문이다.

본 절(節)에서는 본 연구의 배경이 되는 네트워크로 연결된 워크스테이션을 이용한 병렬 프로그램의 동작 환경을 살펴보고, 이러한 환경에서 필연적으로 나타나는 이기종(heterogeneous) 워크스테이션들이 서로 다른 성능을 갖는 것으로 인한 병렬 이상(parallelization anomaly) 현상을 살펴봄으로써 새로운 부하균형방법이 필요한 동기를 서술하며, 이에 관한 그 동안의 관련 연구와 함께 본 논문의 전체구성을 개괄한다.

1.1 병렬프로그램 동작 환경

워크스테이션 클러스터는 본질적으로 재구성이 용이한 상황이다. 즉, 하나의 가상적인 병렬 컴퓨터로서 워크스테이션 클러스터를 구성하는 요소들은 수시로 바뀔 수 있으며 이때마다 응용 프로그램의 구성을 새롭게 작성해야 한다면 이것을 실제적으로 적용하기는 거의 불가능하다. 따라서 본 논문에서 제시하는 부하균형방법은 하드웨어 구성요소의 변화와 그들간의 연결 상태 토폴로지가 변화할 때에도 자동적으로 새로운 구성을 생성해 주는 새로운 프로그램 환경의 존재를 가정하고 있다. 구체적으로 말한다면, 본 논문의 방법은 CORD (Configuration-level Optimization of RPC-based Distributed programs) 환경에서 적용되었으며 CORD는 RPC 기반의 분산 프로그램의 자동 재구성을 위한 프레임워크이다[7]. 이러한 환경에서 프로그램의 성능평가에 영향을 미치는 요소들은 프로그램 구성 단계에서 연결 언어(interconnection language)의 형태로 주석(註釋)이 달린 표현을 이용하여 최적의 message passing primitives를 이용한 분산프로그램을 생성해 낸다. 일반적으로 RPC는 동기화 문제로 병렬 수행이 어렵지만 이러한 제한을 완화한 PARPC 또는 MultiRPC에서 제시된 방법을 이용하여 병렬성을 확보할 수 있다[8,9] 본 논문에서는 이와 같은 기본 환경을 가정하고 이러한 환경에 적용가능한 형태의 부하균형방법을 제시한다.

1.2 연구동기

서로 다른 성능을 갖고 있는 워크스테이션들로 구성된 클러스터에 균등분할방식으로 단순하게 부하균형을 시도하면 병렬처리에 참여하는 워크스테이션들의 수가 늘어나도 수행시간이 줄어들지 않고 더 늘어나는 병렬 이상(parallelization anomaly)이 생길 수 있다. 예를 들어, n 개의 프로세서 집합 $\{P_1, P_2, \ldots P_n\}$이 있을 때 T 개의 균등한 태스크가 있다고 가정하자. 이때 τ_i를 P_i가 단위시간 내에 처리할 수 있는 태스크의 수(數)라고 할 때, 균등 분할인 경우, 각 프로세서는 T/n 개의 태스크를 처리해야 한다. 이때 전체 태스크를 완료하는데 걸리는 시간인 최종수행시간은 가장 작은 τ_i의 값을 (즉, τ_{min}) 갖는 임계성능의 프로세서의 완료시간으로 정의된다. 이 프로세서 집합에 τ_{new} 값을 갖는 새로운 프로세서가 추가되고 이때 τ_{new}의 값이 $n/(n+1)*\tau_{min}$보다 작다면 $(n+1)$ 개의 프로세서 집합으로 구성된 클러스터가 동일 태스크 집합을 완료하는데 걸리는 시간은 $T/((n+1)*\tau_{new})$가 되어 n 개의 프로세서 집합으로 구성된 클러스터의 경우보다 더 느리게 완료된다. 이것은 심각한 병렬이상 현상이며 적절한 부하균형 방법의 도입을 필요로 하는 강한 동기를 제공하고 있다.

위와 같은 병렬 이상 문제를 제거하기 위해 태스크를 각 프로세서의 알려진 기본 성능에 따라 비례 배분하는 방법이 연구되었다[10,11]. 하지만 이 방법은 프로세서의 성능을 정적으로 판단하며 따라서 실제 적용에 큰 효과를 기대할 수 없다. 동적인 loop 스케줄링 방법은 좀 더 일반적인 경우를 취급할 수 있지만 이러한 방법의 중앙집중적인 특성은 중앙에 위치한 프로세서에 병목이 발생할 가능성이 있으므로 또다른 문제가 발생하게 된다.

예를 들어 100개의 서버가 있고 하나의 master 서버가 태스크를 준비하고 보내는데 1/100 초가 필요하다면 각 참여 서버의 평균 처리 시간이 1초 이내인 경우에 병목으로 작동한다. 본 논문 후반부의 Mandelbrot 집합을 이용한 우리의 실험에서는 400 x 400 픽셀의 윈도우 크기를 갖는 프로그램을 Self-Scheduling의 방법으로 수행해 본 결과, 이러한 문제가 발생하지 않으려면 서브 태스크의 크기가 통신 오버헤드에 비해 충분히 큰 값을

가져야 만 한다는 사실을 발견할 수 있었다. 그러나 실제 상황에서는 놀라울 정도로 병렬성이 높은 ("embarrassingly parallel") 응용이 많이 있으므로 이러한 가정은 적절치 않으며 반드시 적절한 형태의 분산된 부하균형 방법이 요구된다고 볼 수 있다.

1.3 관련연구

만일 *I*개의 균등한 태스크가 있고, *P*개의 동일한 성능을 갖는 프로세서가 있다면 부하균형은 단순히 각 프로세서에게 *I/P*개의 태스크를 분배함으로써 얻어지게 될 것이다. 하지만 각 태스크의 작업요구량과 각 프로세서의 처리능력은 미리 알 수 없는 경우가 많이 있으며, 본 논문의 후반부에서 예로 든 바와 같이 본질적으로 사전 파악이 불가능한 경우도 있다. 만일 이와 같은 경우라면, *I/P*와 같이 정적(靜的)인 방법으로 태스크를 나누어서 부하균형에 도달하기는 거의 불가능하다. 이에 대한 가장 간단한 형태의 개선안은 *Self-Scheduling* (SS)이라는 부하균형을 위한 동적(動的) 해결책이다[5]. 이 방법에서는 프로세서가 할당 가능한 상태가 되었을 때에만 하나의 루프 마디를 할당한다. 하지만 여기에는 동기화에 따른 상당히 큰 오버헤드가 필요하다. *Uniform-sized Chunking*(or Chunk Self-Scheduling, CSS)은 SS에서 루프 마디를 하나씩 보내는 대신 *K*개씩 동시에 프로세서로 보냄으로써 동기화에 필요한 오버헤드를 줄이는 방법이다[3]. 여기서 오버헤드는 *1/K*로 줄어들게 되므로 *K*의 값이 커질수록 오버헤드는 작아지지만 부하 불균형은 심화될 수 있다. *Guided Self-Scheduling*(GSS)은 고정된 chunk size *K*를 사용하지 않고 프로그램의 후반으로 갈수록 비선형적(non-linear)으로 줄어드는 값을 갖도록 함으로써 프로그램의 초반에 오버헤드를 줄이고 프로그램의 후반에 부하 불균형을 줄일 수 있는 방법을 제시하고 있다. *Trapezoid Self-Scheduling*(TSS)은 GSS와 비슷한 방법이지만 비선형적으로 *K* 값을 조정하는데 필요한 스케줄링 오버헤드를 줄일 수 있도록 선형적인 감축 함수를 이용하여 *K* 값을 결정하도록 한다[6]. 이렇게 하면 이론적으로 밝혀진 최소의 동기화 오버헤드와 최적의 부하 균형을 얻을 수는 없지만 *K* 값을 결정하는데 필요한 스케줄링 오버헤드를 줄임으로써 결과적으로 더 좋은 효과를 볼 수 있도록 하는 방법이다. GSS 방법을 좀 더 일반화한 방법으로서 chunk의 크기를 참여하는 프로세서에 따라서 factoring 하면서 스케줄링하는 FAC2 기법이 IBM의 shared memory 형태의 병렬 컴퓨터를 위한 컴파일러에 적용되어 GSS 보다 좋은 성능을 보인다는 것이 보고되기도 하였다[1].

위에서 소개된 부하균형방법은 모두 병렬컴퓨터를 위한 컴파일러에서 작업을 균등분할하기 위해 사용하는 방법들이다. 최근에는 네트워크로 연결된 워크스테이션(또는 PC)들을 하나의 가상적인 병렬 컴퓨터로 보고 여기에 병렬 프로그램을 수행하려고 하는 Peer-to-Peer 컴퓨팅에 대한 새로운 시도가 행해지고 있는데 이를 위해 기존의 shared memory 방식 병렬컴퓨터에서 사용된 기법을 그대로 적용할 수는 없다. 본 논문에서는 이러한 일반적인 병렬 프로그램을 최근의 peer-to-peer 컴퓨팅 환경에서도 효율적으로 수행할 수 있도록 하기 위한 기본적인 부하 균형 방법을 제시하고자 한다. 본 논문에서 제시된 방법은 위에서 살펴본 기존의 방법과는 달리 네트워크에 연결된 워크스테이션들의 기본 성능과 태스크들의 불규칙한 작업량을 동시에 고려하여 부하균형을 유지하고 궁극적으로 최종수행시간을 최소화하는 새로운 방법이다. 이를 위해 기본적인 부하균형 기법과 함께 기존에 고려하지 않았던 성능균형 방법을 동시에 적용함으로써 기존의 방법에서 추가적인 성능향상이 가능하도록 한다.

1.4 전체구성

2절에서는 병렬 루프에 관한 몇가지 모델을 상정하고 이것을 클러스터 모델에 적용할 때 요구되는 부하균형 조건에 대해 살펴본다. 3절에서는 클러스터 모델을 엄밀하게 살펴본 후 이러한 클러스터를 어떻게 구성하면 효율적인 토폴로지가 될 수 있는 지를 파악해 본다. 이렇게 얻어진 토폴로지는 태스크의 이동 경로를 최적의 형태로 제공한다는 것을 보일 수 있다. 4절에서는 이러한 새로운 방식으로 구성된 클러스터 모델의 성능 평가를 위한 중요한 부하균형 행태를 수학적으로 모델링하여 분석한 후 5절에서 심각하게 불규칙적인 부하량을 갖는 태스크들로 구성된 응용 프로그램에 대해 본 방법을 적용한 실험 결과를 제시하고 6절에서 결론을 맺는다.

2. 병렬 루프와 워크스테이션 클러스터 모델

본 연구에서는 전형적인 병렬 루프를 그림 1과 같은 4가지 형태로 구분한다. 이러한 반복 주기의 패턴은 부하 균형 방법을 적용하는데 고려해야 할 점을 시사해 주고 있다. 이러한 병렬 루프의 모델을 살펴본 다음 서로 다른 성능을 가진 이기종 워크스테이션으로 구성된 클러스터 모델을 살펴본다.

2.1 Loop 모델

그림 1에서 *L(i)*는 *i* 번째 loop 마디를 계산하기 위한 작업량을 의미하며 이 그림은 각각의 마디가 서로 독립적으로 계산할 수 있는 병렬 루프의 4가지 형태를 보여주고 있다. 첫 번째 그림 (a)는 작업량이 loop가 진행되어도 변하지 않는 경우이다. 그림 (b)와 (c)는 작업량이 마디별로 균등하지는 않지만 선형적으로 증가하거나 감소하는 형태를 취하고 있고 그림 (d)는 완전히 불규칙

적인 양상을 보이고 있다. 처음 3개의 모델은 기본적인 Self-scheduling의 단점을 보완한 기존의 loop 스케줄링 기법[2,4,6]으로 충분히 효율적으로 대처 가능한 병렬 루프들이다.

마지막 형태인 불규칙적인 모델은 다시 예측가능한 경우와 그렇지 않은 경우로 구분할 수 있다. 예를 들어, DNA 염기 서열 유사성을 찾는 응용에서의 불규칙성은 어느정도 예측가능한 형태로 볼 수 있지만 Mandelbrot 집합의 계산은 예측불가능한 경우에 해당한다. 이 모든 경우를 포괄할 수 있는 부하 균형 방법을 생각할 수 있어야 한다.

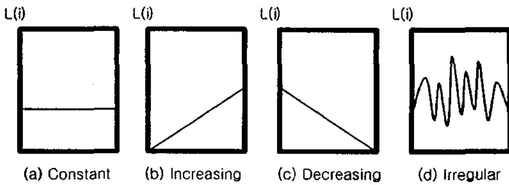


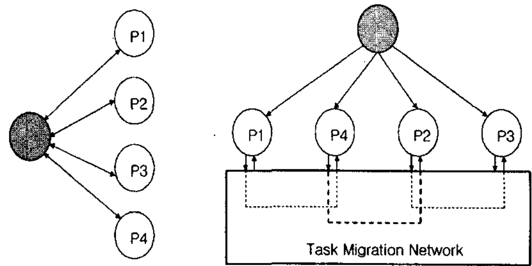
그림 1 독립적인 병렬 루프의 전형적인 4가지 형태

2.2 워크스테이션 클러스터 모델

그림 2는 병렬 루프를 처리하기 위한 워크스테이션 클러스터 모델의 가장 기본적인 두가지 형태의 토폴로지를 보여주고 있다. 그림 2의 (a)는 참고문헌[3-6]에서 소개된 전통적인 loop 스케줄링 기법이 적용되는 프로세서간 연결 구조이다. 이러한 형태에서 이미 프로세서에 할당된 태스크의 추가적 부하균형 노력을 위한 재배치는 고려되지 않는다. 스케줄링 프로세스는 중앙에 위치한 master 프로세서에 집중되어 있으므로 앞에서 언급한 병목현상이 발생할 위험이 있는 형태이다.

그림 2의 (b)는 본 연구에서 제안한 부하균형방법에서 적용되는 연결 구조이다. 전통적인 방법에서와 마찬가지로 master 프로세서가 초기 태스크 할당을 담당한다. 하지만 이미 할당된 태스크라고 할지라도 전체 프로세서의 동적인 부하 상황에 따라 다른 프로세서로 재배치될 수 있는 태스크 이동용 네트워크가(migration network) 추가된 형태이다. 부하균형 유지를 위한 태스크의 동적 재배치는 오직 미리 준비된 이동용 네트워크 경로를 통해서 이루어 진다. 이러한 경로를 미리 마련해 놓은 이유는 동적 재배치 필요성 발생시 자기에게 할당된 태스크를 이전할 프로세서를 찾는 스케줄링 오버헤드를 줄이는 데 유용하다. 본 논문에서 사용되는 클러스터 모델은 다음과 같은 방법으로 요약된다.

- N: 클러스터 내의 워크스테이션의 수,{W_1, W_2, ..., W_N}
- τ_i: W_i의 처리능력(throughput)인 단위 시간내 처리 가능한 단위 태스크의 개수로 환산된 값



(a) 전통적 loop 스케줄링 구조 (b) Task migration Network를 이용한 구조

그림 2 부하균형을 위한 2가지 토폴로지

- γ_{ij}: 프로세서 i에서 j로 재배치되어야 할 부하(workload)의 양

3. 새로운 부하 균형 유지 방법

동적 부하 균형 기법에서 중요한 두가지 요소는 이동정책(transfer policy)와 위치 정책(location olicy)이다. 이동 정책은 부하상태를 고려하여 하나의 태스크가 그냥 현재 배당된 프로세서에서 처리되어야 할지 또는 다른 프로세서로 이동되어 처리되어야 할 지를 결정하는 원칙에 관한 정책이다. 위치 정책은 만일 부하불균형이 발생하였다고 판단되어 이동을 시킬 경우 태스크 이동의 시작지(source) 프로세서와 목적지(destination) 프로세서를 결정하는 정책이다. 운영체제 전체의 측면에서 이와같은 결정을 하려면 시스템의 부하상태에 대한 다차원적인 부하벡터 정보가 필요하지만 우리는 하나의 커다란 병렬 프로그램을 대상으로 부하균형을 시도하고 있으므로 태스크의 크기로 단순화 된다. 따라서 부하상태교환[12] 방법이나 후보 프로세서 결정을 위한 무작위 polling[13]과 같은 복잡한 방법이 필요없고 단지 태스크의 수가 유일한 부하상태를 결정하는 요소이다. 따라서 이동 정책은 부하가 부족한(underloaded) 프로세서가 이미 설정된 태스크 이동용 네트워크를 따라 "demand" 신호를 보내는 것으로 간단하게 결정된다.

위치 정책 역시 이미 이동용 네트워크가 결정되어 있으므로 동적으로 재검색할 필요가 없어서 부하균형 노력에 따른 추가적인 오버헤드를 줄일 수 있다. 사실상 위치 정책은 부하균형 방법의 성능을 결정하는 매우 중요한 요소이다. 참여하고 있는 프로세서의 수가 많을 경우 최적으로 source와 destination 프로세서를 결정하는 것은 상당한 오버헤드를 요구하기 때문이다. 본 연구에서는 이것을 미리 결정된 migration network로 대체하여 부하균형 오버헤드를 최소화 할 수 있다는 것을 이론적으로 계산하고 또 실험결과로 보여준다. 각 프로세서의 순간적으로 변하는 부하 상태는 미리 예측할 수 없지만 각 프로세서의 기본 성능은 이미 알려져 있기

때문이다. 워크스테이션 클러스터를 만들어 사용하는 경우는 대부분 시스템이 오랜 기간 사용되지 않을 때가 대부분이므로 이러한 기본 성능이 결국 동적인 성능과 큰 차이가 없게 된다.

본 절(節)에서는 그림 2의 (b)와 같은 task migration network을 구성하는 방법을 설명한다. 이 그림에서의 점선을 통해 과부하/저부하 프로세서간에 태스크 교환이 이루어진다. 예를 들어, ($P_i \rightarrow P_j$)은 부하균형을 위한 태스크 이동의 기본 단위이다. 빠른 성능을 가진 P_j가 자신에게 할당된 태스크 처리를 끝내고 추가적인 태스크를 P_i로부터 전달받는 상황을 묘사한다. 이때 두 프로세서간 상대적 성능차이에 따라 미리 결정된 γ_{ij} 만큼의 태스크가 이동된다. 그림 3은 PVM을[14] 이용하여 이러한 상황을 서술하는 병렬 프로그램의 예를 보여준다(이러한 프로그램은 자동으로 생성 가능하며 그 생성 방법은 본 논문의 서술 범위를 넘는다).

```
/*P_i:sender*/
for(i=0;i<taskcnt;i++){
     if(pvm_nrecv(P_j,MoreTaskReq)){
          /*a request arrived*/
          n=(taskcnt-i+1)*Ratio_ij
          /*Migrate to P_j*/
          if(n){
             pvm_initsend(PvmDataDefault);
             pvm_pkint(&n,1,1);
             pvm_pkint(&TaskQ[i],n,1);
             pvm_send(P_j,TaskMigrating);
             i+=n;
             continue;
          }
     }
     /*loop body on TaskQ[i]*/
}
/*P_j: receiver*/
LOOP:
     for(i=0;i<taskcnt;i++){
          /*loop body on TaskQ[i]*/
     }
     /*Check the partner processor P_i*/
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&more,1,1);
     pvm_send(P_i,MoreTaskReq);
     /*wait until killed by parent*/
     while(1)
          If(pvm_nrecv(P_i,TaskMigrating)
     /*migrated tasks arrived*/
     pvm_upkint(&taskcnt,1,1);
     pvm_upkint(TaskQ,taskcnt,1);
     goto LOOP;
}
```

그림 3 Task migration path를 포함하여 생성된 병렬 프로그램

먼저 2.2절에서 설명된 클러스터 모델에 대한 정확한 정의를 부여한 후, 이 정의에 기반한 task migration network 형성 방법에 대해 논한다. 모든 클러스터는 (W_s, W_f)의 쌍으로 재귀적으로 정의한다. 즉, 클러스터의 구성요소인 하나의 워크스테이션은 그 자체가 또다시 워크스테이션 클러스터로 구성 가능하다. 여기서 W_s는 클러스터를 구성하는 두 개의 워크스테이션 중 느린(slower) 프로세서를 나타내고, W_f는 더 빠른(faster) 프로세서이다. 이 논문에서는 클러스터 표현을 처리량으로 표현한 (τ_s, τ_f)와 상호교환적으로 사용하기로 한다. 전체 워크스테이션 클러스터는 다음과 같이 정의된다.

정의 3.1 클러스터 트리(cluster tree)

N개의 워크스테이션 {W_1, W_2, ..., W_N}으로 구성된 클러스터 트리(CT)는 다음과 같은 이진 트리 CT = (V, $E_{left} \cup E_{right}$)로 정의된다.

1. 꼭지점 V는 각각 클러스터이다. 특별한 꼭지점인 "root"는 전체 클러스터를 의미한다.
2. E_{left}는 왼쪽 서브 트리이고 E_{right}는 오른쪽 서브 트리이다. E_{right}에 해당하는 클러스터는 E_{left}에 해당하는 클러스터보다 더 빠르거나 같은 성능을 갖고 있다.
3. 두 노드 사이의 태스크 이동 경로는 E_{left} 트리의 최우단(rightmost) 노드를 v라고 하고 E_{right} 트리의 최좌단(leftmost) 노드를 w라고 할때, v에서 w로 설정된다.

정의 3.2 균형 비율(balance ratio)

클러스터 C_1 = (τ_1, τ_2)에서 균형 비율 B_{C_1}은 ($\tau_2 - \tau_1$)/($\tau_2 + \tau_1$)로 정의된다. 만일 이 값이 또다른 클러스터 C_2 = (τ_3, τ_4)의 균형 비율 B_{C_2}보다 더 작으면 클러스터 C_1은 C_2보다, 더 균형적(more balanced)이라고 말한다.

정의 3.3 클러스터의 결합 성능(combined performance)

클러스터 C_1 = (τ_1, τ_2)과 C_2 = (τ_3, τ_4)에서 τ_{C_1}이 τ_{C_2}보다 더 크면 클러스터 C_1이 C_2보다 더 빠르다(faster)라고 말한다. 만일 τ_{C_1}과 τ_{C_2}의 값이 같으면 정의 3.2의 균형비율을 비교하여 더 균형적인 클러스터가 더 빠른 것으로 정의한다.

τ_1과 τ_2가 같은 값을 가질 경우 균형비율은 0이 되며 이것은 두 워크스테이션의 성능이 동등하여 동일한 작업량을 갖는 태스크가 할당되었다면 완벽하게 부하균형이 (perfectly balanced) 이루어짐을 의미한다. 극단적으로 $\tau_1 << \tau_2$인 경우에는 균형비율은 1에 수렴하게 되며 대부분의 태스크가 τ_2의 처리량을 갖는 프로세서에서 수행됨을 의미한다. 만일 초기 부하가 균등 할당되

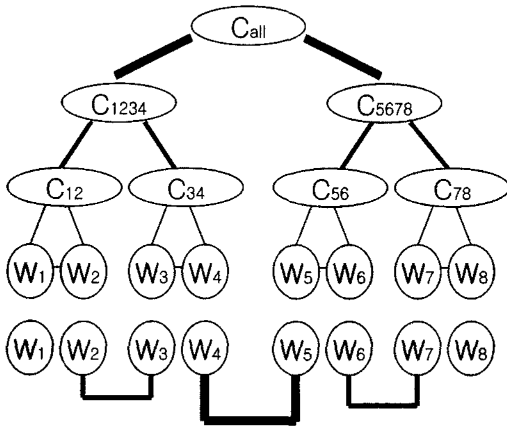


그림 4 클러스터 트리와 task migration path

었고 클러스터 또한 완벽하게 균형을 이루었다면 클러스터 내부에서의 태스크 이동의 가능성은 이론적으로 존재하지 않는다. 즉, 더 균형이 이루어진 클러스터일수록 이동되어야 할 태스크의 양이 적어지게 된다. 이와 같은 상황에서 task migration network은 다음과 같은 방법으로 구성된다. 그림 4는 클러스터 트리와 클러스터를 구성하고 있는 프로세서들 간의 task migration path가 아래의 **알고리즘 1**에 따라 형성된 모양을 나타내고 있다.

알고리즘 1 *Task migration network from CT*

Begin
For all clusters (non-terminal nodes) *c* in *CT*
For two children *v* and *w* such that (*c*, *v*) ∈ E_{left} and (*c*, *w*) ∈ E_{right}
if (*v*, *w* are terminals)
then *CONNECT v TO w*
else *CONNECT* RightmostTerminal(*v*) *TO* LeftmostTerminal(*w*)
End

4. 부하 균형 유지를 위한 태스크 이동 행태 분석

참고문헌 [13]에 따르면 부하 균형 기법에서 중요한 두가지 요소가 있다. 첫째는, 부하 균형 유지로 인한 오버헤드가 부하 균형에 의한 긍정적인 효과를 상쇄해서는 안된다. 둘째로는 두 프로세서간에 비정상적으로 계속하여 서로간에 태스크를 이동시키는 불안정한 부하 균등 현상이 생기지 않도록 해야 한다. 본 논문에서 제시하는 방법은 하나의 프로세서가 처리할 태스크를 다 소모하였을 경우에만 태스크 이동이 발생하므로 불안정성 문제는 존재하지 않는다. 본 절에서는 간단한 예를 통하여 부하 균형을 위한 태스크 이동 행태를 파악해 본다.

N개의 균등한 태스크를 수행할 *P1, P2, P3, P4*로 구성된 클러스터가 있다고 가정한다. 이 프로세서들의 처리량은 미리 알려져 있으며 각각 τ, 2τ, 3τ, 4τ이다. *P1*은 가장 작은 처리량을 가지고 있으므로 태스크가 균등 분배되었다면 태스크를 방출해야 하는 프로세서가 될 가능성이 제일 높다. 이때 방출되는 태스크를 전달받아야 하는 프로세서 후보로 *P2, P3*보다는 가장 처리량이 큰 *P4*가 적당하다. 일반적인 부하 균형 방법에서는 위치 정책에 의거하여 이것을 발견하기 위한 부하 정보 교환을 해야 하지만 이것은 놀고 있는 워크스테이션을 이용한 클러스터에서는 불필요한 작업이다. 따라서 **알고리즘 1**에서는 처리량의 차이가 가장 큰 두 개의 프로세서를 묶어서 하나의 클러스터로 만들었으며 이것은 남는 것과 모자라는 것을 평균하여 부하 균형을 이루는 효과를 나타낸다. 또한 이러한 묶음은 부하 불균형이 발생하여 태스크 이동이 필요할 경우, 이동되어야 하는 태스크 양을 더 크게 할 수 있음으로 데이타 전송시 latency를 줄일 수 있다는 장점도 제공한다.

결과적으로 이 경우에는 (*P1, P4*)와 (*P2, P3*)의 클러스터가 형성된다. 이 두 개의 클러스터는 또다시 하나의 클러스터로 묶일 수 있는데 이 경우에 두 클러스터의 처리량의 합은 같다. 하지만 (*P2, P3*)의 부하균형정도가 (*P1, P4*) 보다 더 높으므로 더 많은 태스크를 처리할 수 있다. 따라서 최종적으로 얻어지는 클러스터는 ((*P1, P4*), (*P2, P3*))가 된다(**정의 3.3** 참조).

물론 위와 같은 실제로는 위와 같은 이상적인 경우가 발생하기 어렵다. 수행 도중 각 워크스테이션의 소유자가 시작한 작업으로 처리능력이 줄어들 수도 있고 무엇보다도 불규칙한 루프 모델(그림 1(d))인 경우 태스크의 "균등한" 배정이라는 것 자체가 가능하지 않다. 이 경우 초기부터 부하는 균등하지 않으며 태스크 이동에 의한 부하 균형이 필수적이다. 본 논문에서 제시한 부하 균형 방법은 (1) 임의의 초기 태스크 배정에도 부하 균형이 이루어지며, (2) 이때 소모되는 스케쥴링 오버헤드는 네트워크 latency time α와 바이트당 transfer time β에 선형적으로 비례하는 함수에 의해 바운드될 수 있고, (3) 이때의 부하균형 결과는 단순히 태스크 처리량에 따른 부하균형이 아닌 모든 참여 프로세서의 종료 시간을 균형시켜서 실제적인 최종완료시간을 단축할 수 있는 실질적인 의미에서의 성능 및 부하균형 방법이다. 클러스터상 가장 멀리 떨어진 두개의 프로세서간에 태스크 이동이 발생할 경우가 최악의 상황이며 이때 p hop이 필요하다고 할 때, 스케쥴링 오버헤드는 $p*(p-1)*(\alpha$

logN + βN)임을 수학적으로 증명할 수 있다. 일반적으로 태스크의 수 N은 홉의 수 p 보다 훨씬 작으며, α는 β보다 훨씬 큰 값임을 기억한다면 이것은 SS 방법만 사용되었을 경우의 오버헤드인 pN(α + β)보다 훨씬 향상된 것임을 이론적으로 보여준다. 다음 절에서 이에 대한 실험결과를 보고한다.

5. 실험결과

본 연구에서 제시한 부하 균형 효과를 살펴보기 위해 우리는 16개의 워크스테이션이 LAN으로 연결된 상황에서 PVM을 이용하여 실험하였다. 불규칙 루프 모델을 갖는 예제로 [0.5, -1.8] × [1.2, -1.2] Mandelbrot 집합을 800 × 800 픽셀을 갖는 그래픽 윈도우에 계산하는 응용 프로그램을 사용하였다. 이 프로그램은 그림 5와 같은 불규칙한 작업량을 갖는 태스크들로 구성되어 있다.

루프 마디들 간의 작업 요구량에 매우 큰 차이가 있고, 이러한 차이는 점진적으로 발생함으로 우리는 태스크를 round robin 방식으로 초기 배정하였다. 사용한 각 워크스테이션의 기본 성능은 그림 6에 나타나 있다. 이 그림은 동일한 프로그램을 네트워크에서 제거한 단독 워크스테이션에 수행하였을 때 최종완료시간을 기록

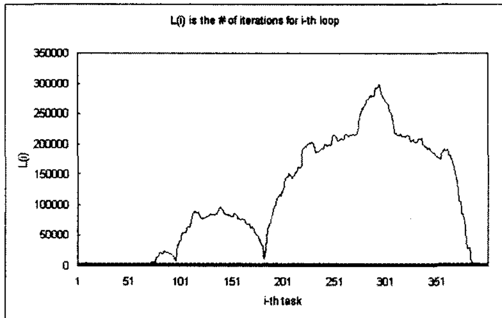


그림 5 Mandelbrot 집합 계산에서 루프 마디별 작업 요구량

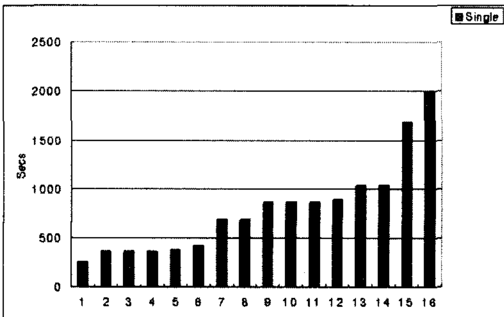


그림 6 16개 워크스테이션의 기본 성능 비교

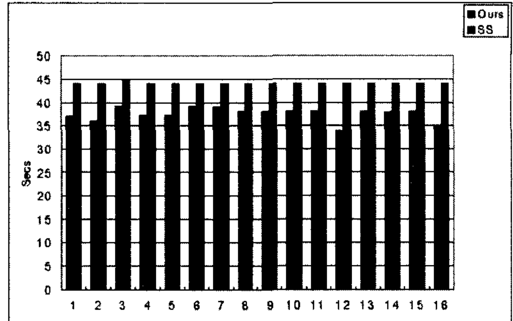


그림 7 새로운 방법으로 연결된 클러스터의 부하균형 결과

한 것이므로 각 워크스테이션의 기본 성능을 나타낸다고 볼 수 있다. 즉, 가장 빠른 성능의 워크스테이션은 본 작업을 완료하기 위해 약 250초 정도의 시간이 걸렸으며 가장 느린 워크스테이션은 2000초 정도 걸렸으므로 최대 약 10배정도에 해당하는 성능의 차이를 갖는 16개의 워크스테이션들이 참여한 클러스터의 부하 균형 성능을 실험하는 상황이다.

16개의 워크스테이션들이 클러스터로 사용되어 Mandelbrot 집합을 계산하는데 걸리는 시간은 그림 7에 나타나 있다. 이 그림에서는 본 논문에서 제시된 새로운 부하 균형 방법과 전통적인 SS 방법이 비교되었다. 이 경우에는 최종수행시간 측면에서 약 25% 정도의 성능 향상을 측정할 수 있었다. 본 그림에서 특이한 사항은 SS 방법은 훨씬 더 정교한 부하 균형 결과를 가져왔다는 것이다. 즉, 16개 워크스테이션의 최종 수행 결과가 거의 일치한다. 우리의 새로운 방법은 비록 부하 균형의 공평성 측면에서는 SS 방법보다 좋지 않지만 (사실상 SS는 가장 완벽한 상태의 부하균형을 항상 유지한다. 문제는 이러한 완벽한 부하 균형을 이루기 위해 필요한 오버헤드가 워크스테이션 클러스터의 경우 너무 과도하다는 데에 있다) 최종 수행 완료 시간은 훨씬 더 일찍 끝나는 것을 관찰할 수 있다.

6. 결 론

본 연구에서 제안한 재구성 가능한 분산 부하 균형방법은 이기종 성능을 갖는 워크스테이션들로 구성된 클러스터에서 불규칙적인 병렬 루프를 갖는 응용을 수행하는데 매우 효과적인 방법이다. 본 논문에서는 전통적인 동적 부하 균형 방법이 불규칙한 특성을 갖는 병렬 루프에서는 적합하지 않은 이유를 논의하였으며 특별히 SS 방법과 그것을 개선한 CSS, GSS 및 TSS는 모두 shared-memory 병렬 구조에 적합한 방법이므로 워크스테이션의 네트워크 클러스터와 같은 통신 오버헤드가

몇십배나 더 큰 상황에서는 병목 현상을 회피할 방법이 없으므로 적용 불가능함을 지적하였다. 이제까지의 부하 균형 방법들이 모두 처리해야 할 대상으로서 태스크의 균등한 배정 및 재배정 방법에만 초점을 맞추었지만 (load balancing) 본 논문에서는 태스크를 처리하는 프로세서들의 처리량에 기반한 성능균형(performance balancing) 방법을 제시하였다는 점에서 큰 의미가 있다. 이러한 성능균형 노력은 부하균형 노력과 결합하면 이동되어야 할 태스크의 수를 줄이고 부하 균형에 도달하는 시간을 단축할 수 있는 장점을 제공하며 이러한 내용은 실험으로 입증되었다. 본 논문에서는 일반적인 트리 구조에 의한 성능 균형 노력만을 살펴 보았는데 앞으로 새로운 토폴로지에 의한 성능 균형 효과를 더 살펴볼 필요가 있다.

참 고 문 헌

[1] S. F. Hummel, E. Schonberg, and L. E. Flynn, Factoring: A method for scheduling parallel loops, Communications of the ACM, Vol. 3(8), August, 1992.

[2] M. Cierniak, W. Li and M. J. Zaki. Loop scheduling for heterogeneity, In Proceedings of the 4th International Symposium on High Performance Distributed Computing, August 1995.

[3] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors, IEEE Transactions on Software Engineering, Vol. 11(10): 1001~1016, October 1985.

[4] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers, IEEE Transactions on Computer, Vol. C-36(12): 1425~1439, December 1987.

[5] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In Proceedings of International Conference on Parallel Processing, August 1986.

[6] T. H. Tzen and L. M. Ni. Dynamic loop scheduling for shared-memory multiprocessors. In Proceedings of International Conference on Parallel Processing, August 1991.

[7] T.-H. Kim and J. M. Purtilo. Configuration-level Optimization of RPC-based Distributed programs. In Proceeding of the 15th Internatinal Conference on Distributed Computing Systems, May 1995.

[8] B. Martin, C. Bergan, and B. Russ. PARPC: A system for parallel remote procedure calls. In Proceedings of the International Conferences on Parallel Processing, 1987.

[9] M. Satyanarayanan and E. H. Siegel. MultiRPC: A parallel remote procedure call mechanism. Technical Report CMU-CS-86-139, Carnegie-Mellon University, 1986.

[10] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments, Parallel Computing, Vol. 19: 1221~1234, 1993.

[11] A. S. Grimshaw, J. B. Weissman, E. A. West and Jr. E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing, Vol. 21: 257~270, 1994.

[12] P. Krueger and N. G. Shivaratri. Adaptive location policies for global scheduling. IEEE Transactions on Software Engineering, Vol. 20(6): 432~444, June 1994.

[13] Derek L. Eager, Edward D. Lazowska, and John Zarhorjan. Adaptive load sharing in homogeneous distributed systems, IEEE Transactions on Software Engineering, Vol. 12(5): 662~675, May 1986.

[14] V. S. Sunderam. PVM: A framework for parallel distributed computing. Concurrency: Practice and Experience, Vol. 2(4): 315~339, December 1990.

김 태 형

1986년 2월 서울대학교 컴퓨터공학과 졸업. 1988년 2월 서울대학교 컴퓨터공학과 석사. 1996년 9월 University of Maryland College Park, Dept. of Computer Science 박사. 1988년 3월~1990년 8월 현대전자산업(주) 응용 SW 개발팀. 1996년 10월~1999년 3월 현대정보기술(주) 정보기술연구소 책임연구원. 1999년 3월~2001년 2월 한양대학교 전자컴퓨터공학부 전임강사. 2001년 3월~현재 한양대학교 전자컴퓨터공학부 조교수. 관심분야는 분산병렬컴퓨팅, 이동컴퓨팅, 내장형 시스템, 미들웨어 시스템소프트웨어 구조