이기종 병렬 시스템을 위한 자동적 병렬화 컴파일러 후위

(Backend of a Parallelizing Compiler for an Heterogeneous Parallel System)

권 대 석 † 김 흥 환 †† 한 상 영 †††

(Daesuk Kwon) (Heunghwan Kim) (Sangyong Han)

요 약 고전적 시스템의 성능 향상을 위해 많은 병렬 처리 시스템들이 제안되어 왔다. 그러나 이들 시스템들은 흔히 통신과 동기화 부담을 과소 평가함으로써 기대한 만큼의 성능을 보이지 못하였다. 본 논문에서는 그러한 결과를 초래하는 이유를 설명하고, 병렬화 컴파일러가 만족시켜야 하는 성능상의 요구조건을 제시한다. 병렬화 결정은 성능 저하를 피하기 위해 반드시 통신과 동기화 부담(overhead)에 대한 분석에 기초하여 이루어져야 한다. 본 연구진은 이러한 발상을 자동적 병렬화 컴파일러 SUIF에 적용하여 SUIF의 후위를 MPI 함수를 이용하는 새로운 후위로 교체하고, 여기에 병렬화 결정의 타당성을 부담 정보에 기초하여 평가하는 능력을 부여하였다. 새로운 컴파일러 후위는 병렬화 가능한 부분이 명시된 SUIF 중간 코드를, 성능 저하를 초래하지 않으면서 MPI 함수 호출을 포함하는 분산 메모리 구조 병렬 프로그램으로 변환한다.

Abstract Many multiprocessing systems have been developed to exploit the parallelism and to improve the performance. However, the naive multiprocessing schemes were not successful as many researchers thought, due to the heavy cost of communication and synchronization resulting from parallelization. In this paper, we will identify the reasons for the poor performance and the compiler requirements for the performance improvement. We realized that the decisions for multiprocessing should be derived by the overhead information. We applied this idea to the automatic parallelizing compiler, SUIF. We substituted the original backend of SUIF with our backend using MPI, and gave it the capability to validate parallelization decisions based on overhead parameters. This backend converts the intermediate code containing spacification of parallelizable regions into the distributed-memory based parallel program with MPI function calls without excessive parallelization that may cause performance degradation.

1. 서 론

고전적 시스템의 성능 향상을 위해 많은 병렬 처리 시스템들이 제안되어 왔다. 그러나 고전적인 병렬 처리 시스템은 대체로 무시할 수 없는 문제점들로 인해 고전적 컴퓨터 시스템에 대한 완전한 대안이 되지 못하였다. 우선 구현상의 문제로, 모든 병렬처리 시스템들이 만족할만한 성능 향상을 거둔 것은 아니어서, 드물게는 여러 개의 처리 요소를 사용함에도 한 개의 강력한 마이크로프로세서를 사용하는 워크스테이션보다 떨어지는 성능을 보이는 경우도 있다. 이러한 문제는 시스템 설계시 통신과 동기화 부담을 과소평가했기 때문인 경우가 많다. 또 다른 시스템 구현상의 문제로, 대개의 병렬처리 시스템이 동일한 처리 요소들만으로 구성되는 결과, 설계와 구현을 거쳐 시스템이 완성될 즈음에는 시스템 구성에 이용된 처리 요소들은 이미 구형이 되어 버리는 경우가 많다. 이를 다시 신형의 처리 요소로 업그레이드

† 비 회 원 : 서울대학교 전산과학과
hyntel@pplab.snn.ac.kr
†† 정 회 원 : 서원대학교 전산과학과 교수
khh@seowon.ac.kr
††† 종신회원 : 서울대학교 전산과학과 교수
syhan@pplab.snu.ac.kr
논문접수 : 1998년 12월 31일
심사완료 : 2000년 5월 30일

하기 위해서는 전체 처리 요소를 바꾸어야 하므로 막대한 추가 비용이 불가피해진다.

한편 사용자의 입장에서 보다 뛰어난 처리 능력을 얻기 위해 병렬 처리 시스템을 사용하려 할 경우에도 두 가지 문제가 있다. 첫째는, 대개의 병렬 처리 시스템이 단일 프로세서 시스템보다 훨씬 비싸다는 경제적 문제를 해결해야 한다는 것이다. 최근에 주목받고 있는 NOW[1] 등의 워크스테이션 클러스터링(workstation clustering)은 제안된 문제들에 대한 유력한 대안이다. 그러나 병렬 프로그래밍의 경험이 없는 일반 사용자에게는 여전히 병렬 시스템 환경과 거기에서 운영되는 FORTRAN-D[2]나, HPF[3]와 같은 병렬 프로그래밍 언어를 익혀야 한다는 문제점이 남는다. 이러한 문제들은 모두 진입 장벽으로 작용하여 병렬 시스템이 기존 시스템의 완전한 대안으로 자리잡지 못한 원인이 되어 왔다.

SUIF[4]나 Polaris[5] 와 같은 자동적 병렬화 컴파일러는 C나 포트란 등의 상용 순차적 언어로 작성된 프로그램을 입력으로 받아들여 병렬화된 프로그램을 생성한다는 점에서 일반 사용자의 진입 장벽을 해소하리라 생각된다. 그러나 불행하게도 이들 자동적 병렬화 컴파일러는 일반적으로 공유 메모리 구조를 가정하고 프로그램 병렬화를 수행하므로, NOW 등의 분산 메모리 구조에 바로 적용할 수 없다는 문제가 있다. 분산 메모리 시스템을 위해서는 병렬화 컴파일러의 후위(backend)가 메시지 send/receive를 포함하는 형태로 목적 코드를 생성해야 한다.

본 연구에서는 무료로 배포되는, 공유 메모리 구조를 위한 자동적 병렬화 컴파일러 SUIF를, 분산 메모리 구조인 이기종 컴퓨터 네트워크에 이식하였다. 이를 위해 기존의 SUIF 후위 모듈은, 분산 메모리 구조를 가정하고 MPI (Message Passing Interface) 함수들을 첨가함으로써 병렬 프로그램을 생성하는 새로운 후위 모듈로 대치되었다.

단순히 병렬화 가능한 부분이 명시된 중간코드를 분산 메모리 구조에 적합한 병렬 프로그램으로 변환하는 것으로는 충분치 않다. 일반적으로 분산 메모리 구조의 경우, 통신 부담이 공유 메모리 구조의 경우보다 크므로 프로그램의 병렬화 결과로 발생하는 통신 부담이 프로그램의 병렬화로 인해 얻어지는 시간 단축보다 커질 수가 있는 것이다. 따라서 중간 코드로부터 분산 메모리 구조의 병렬 프로그램을 생성할 때 병렬화 가능한 영역 각각에 대해, 분산 메모리 구조에서 추가적으로 발생하는 부담을 평가하여, 병렬화할 것인지 아니면 원래의 순차적 실행을 유지할 것인지 결정해야 한다. 이러한 과정을 거치지 않은 단순한 프로그램 병렬화는 상대적으로 통신 부담이 큰 분산 메모리 모델에서 성능상 저하(performance degradation)를 야기하는 과도한 병렬성(excessive parallelism)을 초래할 수 있다.

애초에 본 연구는 동형의 처리요소들로 구성된 대규모 병렬처리 시스템의 문제를 극복하고, 낙후된 처리요소들까지도 포함하여 보다 높은 성능을 달성할 수 있는 시스템을 구현하기 위해 수행되었다. 어떤 연결망이 그들 사이를 연결하건, 성능상 개선을 얻어낼 수 있다고 판단되기만 하면 병렬화하는 시스템을 구축하려 하였다. 이것은 최근 유행하는 이기종 컴퓨터 네트워크(network of heterogeneous computers)와 결과적으로 비슷한 형태이나, 낙후된 컴퓨터를, 가능한 경우에 성능 개선에 투입할 수 있도록 하자는 연구 동기와 연구의 초점에서 차이를 가진다. 낙후된 컴퓨터와 최신 컴퓨터를 모두 투입, 고성능을 얻고자 한다는 본 연구의 특징은 연구진으로 하여금 이 연구를 patchwork(누더기) 프로젝트라 명명하게 했다. 본 연구진은 이러한 특성의 시스템을 구축하기 위한 기반 환경으로 SUIF 병렬화 컴파일러와 MPI 의 비상업적 구현중 Ohio 대학의 LAM[7]을 선택하였다.

본 연구의 발상과 취지 및 가능성을 검증하기 위해 333MHz 클럭 주파수의 Linux/Pentium-II PC 2대와 300MHz 클럭 주파수의 Linux/AlphaPC 2대를 100 Mbps 이더넷 버스를 이용하여 연결하여 환경을 구축하고 몇 개의 프로그램을 이식하여 성능을 측정하였다.

2절에서는 병렬화 가능한 부분이 명시된 중간 코드로부터 분산 메모리 구조에 기반한 병렬 프로그램을 생성하는 방법에 대해 설명한다. 3절에서는 프로그램 병렬화를 위한 기본적인 컴파일러 요구 조건이 설명될 것이다. 컴파일러 후위를 구현하면서 본 연구진은 병렬화에 대한 결정이 병렬화 대상인 프로그램 분할(partition)들의 실행시간과 병렬화에 수반되는 각종 부담들에 기초하여 이루어지도록 노력하였다. 4절에서는 각 처리요소의 성능, 프로그램 분할의 실행시간, 각종 부담들을 측정하는 전략과 방법 및 몇몇 프로그램에 적용한 결과에 대해 설명한다. 본 실험에는 단지 4대의 컴퓨터가 사용되었을 뿐이지만, 그 결과는 보다 큰 구성에 대해서도 적용가능할 것이다. 마지막 절에서는 향후 연구 방향과 본 시스템의 한계에 대해 논의할 것이다.

2. 분산 메모리 구조를 위한 프로그램 변환 전략

2.1 SUIF와 LAM

SUIF 컴파일러는 미국 Stanford 대학에서 개발된 자동적 병렬화 컴파일러이다. SUIF는 보통의 C 프로그램이나 FORTRAN 프로그램을 읽어들여 병렬화된 중간 코드를 생성한다. 원래 SUIF는 스탠포드 대학 중간 코드 형식(Stanford University Intermediate Format)을 의미하며, 원시 프로그램으로부터 기계어나 어셈블리어로 번역되는 중간 과정에서 이용되는 중간 코드의 표준으로 제시된 것이다. 병렬화된 중간 코드는 다시 C로 변환 가능하며, doall 이나 doacross 등을 포함한다. 이 컴파일러는 많은 중간 단계들로 구성되는데, 각 구성 단계는 각각의 역할을 수행하며 서로 무관한 독립적 프로그램 모듈이다.

일반에 공개된 SUIF 컴파일러는 주어진 원시 프로그램 내에 포함된 모든 루프에 대해 최대한의 병렬성을 찾은 뒤 실행 길이(run-length)는 최대한 길게하면서 통신과 동기화는 가능한 한 최소화하도록 프로그램을 조정한다. 조정된 프로그램은 병렬화 가능한 루프에 대해 '병렬화 가능(doall)'이라 부기(annotation)되어 pgen이라는 프로그램에 의해 최종적으로 병렬화된 중간코드 출력이 생성된다. 이 프로그램은 공유 메모리를 가정한 다중 프로세서 시스템을 위한 병렬 코드를 생성한다.

본 연구의 취지가 임의의 컴퓨터들로 구성된 시스템을 구축하는 것이므로, 이 최종 단계의 프로그램은 분산 메모리를 지원하는 메시지 전송 모델이나 적어도 분산 공유 메모리(distributed-shared memory)를 지원하도록 고쳐지거나 다시 작성되어야 했다. 분산 공유 메모리의 경우라면 기존 pgen에 대해 추가적으로 지원해야 할 사항은 별로 없다. 그러나 이 경우, 시스템은 메모리 접근 시간이 접근 영역에 따라 다른 NUMA(Non-Uniform Memory Access)의 특성으로 인해 성능상 저하를 겪을 수 있다. 이러한 문제들은 해결 가능한 문제이지만, 명시적으로 메시지 전송을 수행하는 메시지 전송 모델이 통신 시간을 측정하는데 있어 분산 공유 메모리보다 유리하다고 생각되었다. 분산 공유 메모리는 사용자에게는 공유 메모리 구조처럼 보이지만 실제로는 숨겨진 수준에서는 메시지 전송을 통해 데이타에 접근을 제공하므로, 어떤 자료에 대한 접근이 얼마만한 처리 시간을 요구할지 예측하거나 측정하고 판단하기가 쉽지 않다. 따라서 데이타가 처리 노드에 존재하는가 아닌가 조사하고, 그 결과와 접근시간들을 기억해 두지 않으면 안된다. 반면에 메시지 전송 모델은 다른 처리 노드가 가진 자료에 접근하기 위해 "명시적으로" 메시지를 주고받으므로 메시지 send/receive가 호출될 때마다 그 소요 시간을 예측하거나 측정하기만 하면 되는 것이다.

오하이오 대학에서 개발된 LAM은 고전적인 메시지 send/receive를 사용하는 표준 MPI를 구현한 것이다. SUIF와 마찬가지로 LAM은 매우 잘 모듈화되어 있으며, 하드웨어와 운영체제 등의 기반과 비교적 무관하게 원만히 잘 동작하는 것을 목표로 하여 구현되어 있다. MPI 표준을 만족시키는 구현으로 MPICH[6] 등이 있으나, 우리는 LAM을 사용하였는데 이는 LAM이 매우 유용한 시각화 도구들과 성능 측정 도구들을 제공하기 때문이었다. 특히 LAM은 워크스테이션 클러스터나 다중 프로세서 시스템을 시스템 구성 요소로 포함할 수 있기 때문에 향후 확장성의 면에서 MPICH보다 유리하다고 판단되었다.

2.2 병렬 프로그램의 생성

프로그램의 실행에 있어서는 같은 프로그램을 모든 노드들이 탑재하여 실행하는 SPMD 모델과, 가장 우수한 성능을 갖는 처리 노드가 작업을 할당하고 결과를 취합하는 마스터-슬레이브(master-slave) 모델을 채택하였다. 개발할 시스템은 다양한 기종의 컴퓨터들로 구성될 것이므로, 그들 가운데 가장 뛰어난 성능의 컴퓨터가 존재하고, 특별히 제작된 병렬 처리 시스템보다 상대적으로 매우 큰 통신 부담이 존재할 것이었다. 따라서 가장 강력한 컴퓨터로 하여금 가능한 한 많은 작업을 처리하도록 하는 것이 유리하리라 판단하였으므로, 마스터-슬레이브 모델이 채택된 것이다. 시스템을 구성하는 모든 처리 노드들은 자기 자신만의 프로그램 복사본을 가지고 실행하지만, 가장 강력한 처리 노드가 초기에 데이타를 나눠주고 다른 노드들로부터 처리 결과들을 취합하게 하였다.

공유 메모리 구조 병렬 시스템과는 달리 분산 메모리 환경의 병렬 시스템에서는 별도의 동기화가 없는 한 모든 프로세서들이 각각의 메모리에 저장되어 있는 프로그램을 독립적으로 실행한다. 본 연구에서는 마스터-슬레이브 모델을 가정하였으므로, 병렬적으로 수행되어야 하는 영역 외의 모든 순차적 작업은 마스터 노드에서만 실행되도록, 자기 자신이 마스터 노드인가 확인하고 작업을 처리하도록 하였다. 생성되는 병렬 프로그램의 기본적 구조는 다음 프로그램과 같다.

각 처리 노드들은 자신이 마스터 노드인지 확인하고 자신이 마스터 노드일 경우 병렬화 하도록 명시된 부분까지의 순차 실행 부분을 처리하다가 병렬 doall 루프를 만나게 되면 doall 서버 함수를 호출하여 모든 처리 노드들이 해당 doall 루프를 실행하도록 하였다. 만일 자신이 마스터 노드가 아니면 모든 노드들은 즉시 doall

```
main()
{
   if (my_node_id == Master_id) {
        원래 코드; (순차 실행 부분)
        call doall_server(doall_loop_id);
        원래 코드; (순차 실행 부분)
        call doall_server(doall_loop_id);
   }              :
   else call doall_server(0);
}

void doall_server(int doall_loop_id)
{
    if (my_node_id == Master_id) {
        MPI_Bcast(doall_loop_id를 브로드캐스팅);
        doall_loop_id에 해당하는 doall_loop을 가진 함수를 호출;
    }
    else {
        while (doall_loop_id >= 0) {
                MPI_Bcast(doll_loop_id를 기다려 받음);
                해당 doall_loop을 가진 함수를 호출;
        }
    }
}
```

프로그램 1 제어구조 변환

서버를 호출한다. doall 서버는 여러 doall 루프들 중, 어느 루프를 수행할 것인가 마스터 노드가 방송(broadcasting)하기를 기다려 해당 doall 루프를 실행하는 함수이다. 각각의 doall 루프는 고유한 id를 할당받고 별도의 함수로 분할된다. doall 서버 함수는 마스터가 전송한 doall 루프 id를 보고 해당 루프를 담고 있는 함수를 호출한다. 만일 마스터가 0보다 작은 id를 방송하면, 프로그램의 끝에 해당되어 모든 처리노드가 수행을 마친다.

공유 메모리 구조에서는 모든 데이타가 모든 프로세서들에 의해 접근 가능하지만, 분산 메모리 구조에서는 데이타를 가지고 있는 노드로부터 데이타를 필요로 하는 노드로 데이타가 명시적으로 전송되어야 한다. 병렬화된 프로그램 분할 각각에서 이용되기는 하지만 그 분할 내에서 앞서 정의되지 않은 데이타들은 그 데이타를 정의한 노드로부터 전송되어야 한다. 이런 기준으로 보아 어떤 데이타가 전송되어야 하는가를 알아내는데 데이타 흐름 방정식(dataflow equation[8])과 SUIF에서 제공하는 DEF, USE 집합을 이용하여 모든 doall 루프를 검사하였다. SUIF가 현재 상황에서는 루프에 대해서만 병렬화를 수행하므로, 프로그램의 모든 코드 블록에 대해 이러한 작업을 할 필요는 없다. MPI는 고전적인 메시지 send/receive 호출을 사용하므로 메시지를 주고받는 함수 호출을 삽입하는 기능을 병렬 코드 생성기인, SUIF의 pgen에 짜넣었다. doall 루프가 각 노드들에 의해 수행을 마친 뒤에 계산 결과는 마스터 노드로 전송되게 하였다. 사실 각각의 노드들이 생성한 결과 모두를 전송하는 것은 필요한 결과만을 전송하는 것보다 많은 통신량을 초래한다. 그러나 본 연구에 이용되는 네트워크가 저가의 TCP/IP 네트워크라서 지연시간과 대역폭이 모두 상대적으로 크므로, 통신량 감소가 성능 향상에 기여하는 이점을 얻기보다는 필요한 자료 모두와 생성한 자료 모두를 전송함으로써 기대할 수 있는 안전성과 통신 회수 감소의 이점을 얻기로 하였다. Ethernet은 TCP/IP에서 약 1ms의 지연시간을 가지므로, 많은 경우에 통신량보다는 통신을 시작할 때 수반되는 지연시간이 통신 부담의 대부분을 차지하는 점을 중시한 것이다.

2.3 워크스테이션 클러스터 상에서의 병렬화 컴파일러

병렬화 컴파일러에 관해서는 SUIF[4], Polaris[5] 등이 대표적이며, 이기종 컴퓨터들을 고속 네트워크로 연결한 시스템의 구현에 관한 연구로는 NOW[1], SHRIMP[10] 등이 있다. 또, 성능 평가에 기초한 실행시간 예측 트레이스 스케줄링에 바탕한 실행시간 최적화[11]와, 통신 시간과 통신 패턴에 따른 프로그램 분할[12] 각각에 대해서도 선행연구가 있으나 본 연구의 근본 취지인 '개별적 신기술을 하나로 모은 시스템'에 대해서는 아직 선행 연구가 미미한 실정이다. 본 연구의 알려진 유사 연구로는 일리노이 대학에서 개발한 자동적 병렬화 컴파일러인 Polaris 컴파일러 후위를 개량하여 MPI 호출을 포함하는 병렬 프로그램을 생성하게 하려는 시도가 있다. 그러나 이 연구는 현재 초기 단계를 수행중이며, 포트란 응용 프로그램을 입력받아 고속 네트워크로 연결된 워크스테이션 클러스터 상에서 수행되는 MPI 병렬 프로그램을 생성하는 연구가 진행되고 있다[13].

3. 병렬화 컴파일러에 대한 기본 요구조건

병렬 프로그램에서 하나의 분할(partition)이란 순차적으로 실행되는 명령어들의 집합으로서, 병렬 수행 환경에서 스케줄링과 실행의 단위를 의미한다. 이러한 실행 단위들을 여러 프로세서에 분산 실행함으로써 성능 개선을 얻게 되는데, 이러한 분할/분산이 어느 때 성능 개선을 가능하도록 하는지 판단해야 한다. 이를 **다중 처리를 위한 요구조건 (requirements for multiprocessing)**이라 부르자.

프로그램을 병렬화하고 병렬수행하기 위해서는 추가적 부담(overhead)들이 발생한다. 순차적 코드에 병렬화를 위해 추가되는 코드의 실행시간, 여러 처리 요소들 간의 통신에 의해 발생하는 통신 지연, 동기화 지연, 처

리 요소들 각각이 겪게 되는 캐쉬 미스(cache misses) 등이 그것이다. 이러한 추가적 성능저해 요소들을 영어 대문자 'O'로 쓰기로 하자. 만일 단일 처리 요소의 전통적 컴퓨터에서 실행한다면 이러한 부담들은 발생하지 않을 것이다. 단일 처리 요소 시스템과 병렬처리 컴퓨터 모두에서 요구되는 순수한 작업 처리 시간을 'R'이라 쓰자.

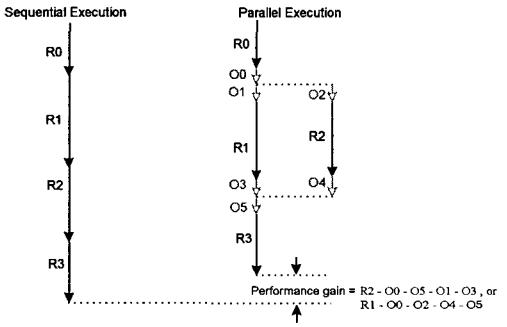

그림 1 병렬 수행을 통해 얻어지는 성능 향상

설명을 쉽게 하기 위해, 2개의 처리요소에 스케줄된 프로그램의 수행만을 생각해 보자(그림 1). 분할 0이, 두 개의 처리요소에서 동시 실행 가능한 두 개의 분할 1과 2에 앞서 수행되어야 하며, 이들이 수행을 마치고 제어가 다른 프로그램 모듈인 3에 의해 통합된다. 물론 분할 0, 1, 2, 3은 각각 여러 프로세스들의 집합일 수도 있다. 이 분할들 각각의 수행 시간을 각각 R0, R1, R2, R3이라 하면, 이들을 단일 처리요소에서 실행할 경우 전체 실행 시간 $R = R0 + R1 + R2 + R3$이 된다.

이를 두 개의 처리요소에 나누어 수행시킬 경우, 불가피하게 추가 부담이 발생한다. 프로그램 코드를 분할하여 전송하는데 필요한 시간이 요구될 수도 있고, 다른 프로세서에 적재된 프로그램 코드를 실행시키거나, 처리할 데이타를 전송하는데 걸리는 시간이 필요할 수도 있다. 이러한 요소들은 R0 뒤에 요구되며, 0번 분할이 분할 1, 2를 생성하므로, 이를 O0라 하고 **생성 부담(spawning overhead)**이라 하자.

생성 부담으로 인한 지연 후, 1, 2번 분할이 실행된다. 0번 분할에서 1번 분할로는 문맥전환 시간이 요구되며, 2번 분할은 실행 준비를 위한 초기 캐쉬 미스 등의 초기 부담을 가질 수 있다. 만일 이들이 공유 자원(메모리 등)으로부터 자료를 적재해야 할 경우, 공유를 위해 요구되는 장치에의 접근 시간이나 처리 시간 등이 추가로 요구된다. 각 분할 1, 2를 위한 이러한 추가 부담을 O1과 O2라 하고, 이를 **준비 부담(preparing overhead)**이라 하자.

O3과 O4는 각 분할 1, 2가 실행 결과를 저장하거나 전송하고 분할 3이 실행되도록 알리는데 필요한 시간 부담을 가리킨다. 이 시간 부담은 **퇴거 부담(retiring overhead)**이라 하자. 이 시간에는 시스템 자원 획득을 기다리는 시간이나 다른 처리 요소의 처리 완료 신호를 기다리는 시간이 포함된다. 한 처리요소에서 다른 처리요소의 처리 완료를 기다려야 하기 때문에 요구되는 시간낭비를 **성능상 낭비된 토막(performance fragmentation)**이라 부르자. 이 명칭은 자료 저장에서 저장할 자료가 저장 단위에 맞지 않아 낭비되는 기억장치(memory fragmentation)로부터 따온 이름이다. 이들은 '*이용될 수도 있으나 이용되지 못한* 낭비라는 점에서 공통점이 있다.

퇴거 부담으로 인한 지연 후, 처리요소는 분할 3을 실행 개시하며, 이에 요구되는 문맥 전환 시간 등을 O5라 하면 병렬 처리시 총 실행 시간 R'는

$R' = R0 + O0 + \mathrm{Max}(O1+R1+O3,\ O2+R2+O4) + O5 + R3.$

따라서 병렬처리를 통한 성능 개선분을 B_{mp}라 하면,

$$\begin{aligned} B_{mp} &= (R0 + R1 + R2 + R3) - (R0 + O0 + \mathrm{Max}(O1+R1+O3,\ O2+R2+O4) + O5 + R3) \\ &= R1 + R2 - O0 - O5 - \mathrm{Max}(O1+R1+O3,\ O2+R2+O4) \\ &= R2 - O0 - O5 - O1 - O3 \quad (O1+R1+O3 > O2+R2+O4\text{의 경우}) \text{ 또는,} \\ &\quad R1 - O0 - O5 - O2 - O4 \quad (O1+R1+O3 < O2+R2+O4\text{의 경우}) \end{aligned} \quad \text{(식 2.1)}$$

어떤 경우에도, 이 값은 0 또는 어떤 임계치보다 커야 한다. 만일 이를 만족시키지 못할 경우, 프로그램 병렬화는 오히려 성능 감소를 야기할 것이다. 물론 이 요구조건은 어떠한 다중 프로세서 시스템에도 동일하게 적용가능하다.

4. 병렬화 유효성 평가의 구현 및 결과

4.1 평가 방법과 컴파일 방법

통신 지연 시간 등의 부담에 관한 정보와 프로그램 분할의 실행 시간을 알아내는 방법은 많다. 클럭 수준의 시뮬레이션은 비교적 정확한 정보를 알아낼 수 있으나 너무 많은 시간이 걸리는 단점이 있다. 시뮬레이션에 의하지 않은 순수한 예측[9]은 결과를 보다 빨리 알아낼 수는 있으나 예측 결과가 정확하지 않다는 단점이 있다. 따라서 여기서는 실제 실행시켜 보고 그 결과값을 이용

하는 방법을 택했다. 그러나 프로그램 전체를 실행해서 시간 관련 정보를 추출하는 것은 매우 비현실적이므로, 최악의 경우를 제외하고는 일부 실행을 통해 정보를 추출하도록 하였다. 새로 설계, 구현된 병렬화 코드 생성기인 pgen은 n개 처리 노드들이 1/n의 작업을 하도록 단순 병렬화(분할)된 코드를 생성하면서 성능 및 부담 관련 정보를 추출하기 위한 테스트 프로그램을 함께 생성한다. 이 테스트 프로그램은 원래 프로그램의 병렬화 결과와 거의 비슷하지만, 세가지 점에서 다르다. 첫째, 테스트 프로그램은 생성 부담, 퇴거 부담, doall 루프 각각의 전후에 시각 측정 함수를 포함한다. 둘째, 테스트 프로그램은 얼마나 많은 각각의 부담과 doall의 소요 시간이 얼마나 되는가 결과를 보고하는 함수를 포함한다. 마지막으로 각각의 처리 노드에 의해 수행되는 doall은 할당된 반복 회수가 5회를 넘으면 5회만 반복한다. 5회라는 회수가 특별한 의미를 갖지는 않는다. 테스트 프로그램의 doall에서 반복 회수는 임의로 정할 수 있다.

이 테스트 프로그램은 LAM 환경하에서 번역되고 실행된다. 실행중에 테스트 프로그램은 통신 부담과 각 doall 루프의 몸체를 1회 실행하는데 걸리는 시간을 모든 처리 노드에 대해 측정하고 화면에 출력해 준다. 출력된 부담 및 실행 시간 정보를 읽어들여 각각의 분할이 병렬적으로 수행되는 것이 성능상 개선을 가져올 수 있을지 여부를 결정하는 프로그램이 별도로 개발되었으며, eval이라 이름지어졌다.

eval의 기능은 매우 단순하다. eval은 전체 노드들 가운데 가장 짧은 실행 시간을 보인 노드를 선택하여 루프 몸체를 1번 실행하는데 필요한 시간에 전체 반복회수를 곱하여 그 노드에서 그 루프가 전부 실행되면 시간이 얼마나 걸릴지 계산한다. eval은 각 노드들의 연산능력을 평가한 후, 각 노드들이 실행할 루프의 하한과 상한을 다시 설정함으로써 각 노드들이 비슷한 시간 내에 작업을 마치도록 작업량을 평준화한다. 그 다음, eval은 평준화된 doall의 실행 시간을 계산해, 그 값과 측정된 생성/퇴거 부담을 처음 계산한 예상 순차 실행 시간에서 뺀다. 만일 이 값이 0보다 작다면 병렬화 부담으로 인해 병렬화는 성능저하를 초래할 것이다. 물론, 각각의 반복이 같은 노드 내에서도 다양한 실행시간을 보일 수 있는데, 이 경우가 앞서 말한 최악의 경우이다. 이런 경우에는 원래대로 전체 반복을 수행하여 실행시간을 측정한다.

eval에 의해 생성된 보고를 이용하여 pgen은 doall을 재분할하여 병렬화된 코드를 생성한다. 재분할에 따라 doall의 하한과 상한, 전송되는 데이타 등이 재조정된다. 표 1은 이 과정을, 표 2는 이상에서 설명한 내용과 반복회수의 조정에 이용되는 식 등을 보여준다.

표 1 성능 지표 추출과 재분할 과정

Tool	작업 내용	작업 산물
SUIF 병렬화 컴파일러	원본 프로그램을 컴파일하고 병렬 루프를 찾아 주석을 첨부함	확장자 .skw SUIF 중간 코드
↓	↓	
dest	.skw 파일을 처리하여 전송될 자료들을 파악, 주석으로 첨부함	확장자 .skx SUIF 중간 코드
↓	↓	
pgen	.skx 파일을 처리하여 프로그램을 단순 분할하고 테스트 프로그램을 생성함	확장자 .sfp SUIF 중간 코드와 확장자 .sfp.test 인 테스트 병렬 SUIF 코드
↓	↓	
scc, s2c, hcc, mpirun	.sfp.test 중간 코드를 C로 바꾸고, C 코드를 컴파일한 뒤 실행	계산 성능 및 통신 성능 지표 정보
↓	↓	
eval	성능지표를 이용하여 병렬화 여부와 병렬화시 노드간 처리 비중 결정	병렬화 결정 타당성 정보와 노드간 비중 정보
↓	↓	
pgen	처리 비중을 이용하여 루프 재분할	재분할된 SUIF 중간 코드

표 2 Evaluated terms for equalizing (load balancing)

T_i	(Excution time of 5-iterations) / 5
P_i	$\frac{1/N_i}{\sum_{k=1}^{N} 1/T_k}$
I_i	$R_{tot} * 1/N$
I_i'	$R_{tot} * P_i$

(R_{tot} : 전체 반복 회수, P_i : node$_i$의 연산능력 , T_i : 노드$_i$에서 1회 반복의 실행시간,
N : 전체 참여 노드의 수, I_i : node$_i$에 할당된 반복 회수 (평준화 이전), I_i' : 재조정된 반복 회수)

4.2 실행 결과

이상의 방법의 유효성을 검증하기 위해 고전적인 행렬 곱셈 프로그램, 병합 정렬, 순회 세일즈맨 문제, NAS 벤치마크 프로그램 등을 실행하였다. 행렬 곱셈은 500 x 500 크기의 실수 행렬 2개를 2중 루프를 돌려 초기화 한 후 고전적인 3중 루프를 돌려 결과 행렬을 계산해내는 순차적 C 프로그램이었다. 이 프로그램의 경우 초기화 루프와 결과 행렬 계산 루프 모두 완전 병렬 루프로서 모두 병렬화 가능하지만, 초기화 2중 루프

는 초기화 결과를 마스터 노드에 반환하는데 걸리는 시간이 초기화를 위해 소요되는 시간에 비해 훨씬 짧으므로 병렬화하면 오히려 성능 저하를 겪게 되고, 결과 행렬 계산의 3중 루프만이 병렬화를 통해 성능 개선을 얻을 수 있다고 추측할 수 있다. 개발된 프로그램들과 컴파일러 후위는 제안된 방식에 따라 평가를 수행하여 이러한 추측대로 결론을 내렸으며, 그 실험 결과는 다음과 같다. 그림에서 Predicted는 eval이 예상한 실행 시간이며, Measured는 실제 실행 시간을 측정한 값이다. 단순 병렬화된 코드와 재분할된 코드는 재분할로 인한 효과를 보여주기 위해 서로 동일한 통신 방법을 사용하였다.

실제 이 프로그램의 실행 결과는 *Pentium-II* 333 MHz 단일 노드에서의 실행 결과가 6.1918초, 컴파일러에 의해 생성된 단순 분할 프로그램의 실행 시간이 5.1882초, 재분할된 프로그램이 3.779162초로 예측 결과와 1% 안팎의 오차를 보였다. 재분할된 코드는 4개의 처리노드를 가지고 1.64배의 성능 향상밖에 보이지 못하였으나 병렬화된 프로그램 수행 시간의 25%가 자료 전송에 소요되는 시간임이 측정되어, 네트워크 성능이 강화되면 충분한 성능 개선을 기대할 수 있다고 기대되었다.

이에 더하여 NAS 벤치마크 프로그램의 순차 프로그램 버전을 입수, SUIF 컴파일러로 병렬화를 실시하였으나 SUIF 컴파일러는 의미있는 병렬성을 찾아내는데 실패하였다. SUIF가 찾아낸 대표적인 병렬 루프는 NAS 벤치마크 패키지를 구성하는 모든 프로그램에서 전형적으로 발견되는 DAXPY 루프였다. 다음 프로그램이 DAXPY 루프의 내용이다.

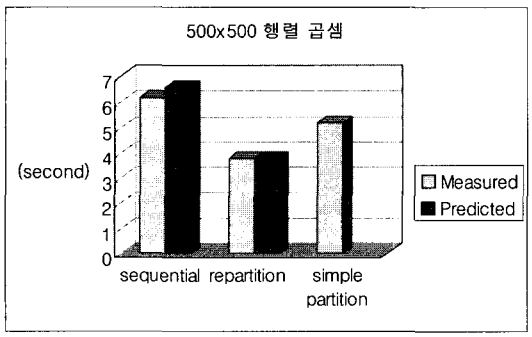

그림 2 500x500 행렬 곱셈 결과

```
double y[N], x[N], a;

for (i = 1; i <= N; i++)
      y[i] = y[i] + a * x[i];
```

프로그램 2 DAXPY code

이 루프는 각각의 반복이 완전히 개별적으로, 상호 작용없이 실행가능하다. 구현된 컴파일러는 이 프로그램에 대해 첫 단계에서 테스트 프로그램과 단순 분할된 루프 프로그램을 생성한다. 생성된 단순 분할 프로그램은 실행 결과 0.461491 초의 실행 시간을 보였는데, 333 MHz Pentium-II PC 단일 노드에서 수행된 결과는 0.005781 초의 실행 시간을 보임으로써 약 8000%의 실행 시간을 갖게 되었다. 이것은 앞서 설명한 과도한 병렬성의 예로, 단순 컴파일러는 이런 경우에 전체 성능을 저하시키는 병렬화를 감행할 것이다. 그러나 본 연구에 의해 구현된 분석기 eval은 테스트 프로그램이 보고한 성능 지표에 대해 분석을 실시하여 이러한 병렬화가 성능상 저하를 초래함을 결론 내리고 최종적으로는 병렬화를 포기하여 원형 코드를 유지하였으며, 병렬화로 인한 성능 저하를 회피할 수 있었다.

순회 세일즈맨 문제는 전형적인 NP-Complete 문제로, 여기서는 가지치기(branch-and-bound) 방법을 이용하여 15개 도시 모두를 순회하고 원점으로 돌아오는 최단 경로를 계산하는 프로그램을 C로 작성하여 병렬화와 재분할을 실시하고 실행시간을 측정하였다.

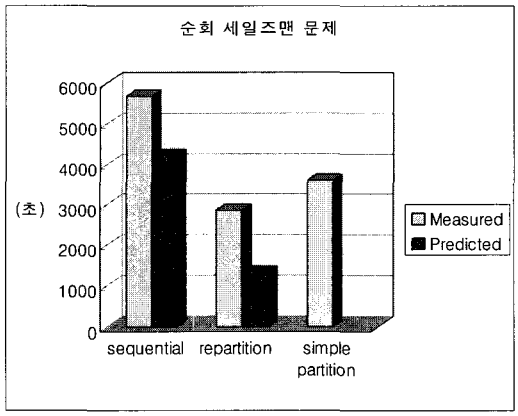

그림 3 순회 세일즈맨 문제 (15도시)

실행 결과 단일 노드에서 소요 시간은 5681.2초, 단순 분할된 병렬 코드에서의 소요 시간은 3607.2초, 재분할된 병렬 코드에서의 소요시간은 2859.99초로 재분할된 코드는 단순 분할된 코드에 비해서는 약 1.3배 더 빠르고, 단일 노드에서 실행된 결과보다는 약 2배 더 빨랐다.

이 코드의 경우는 예측오차가 30% 전후로 상당히 부정확했는데, 이는 각 루프 반복의 실행 시간이 반복마다 다르기 때문이다. 그럼에도 전체 반복중 일부 반복 구간

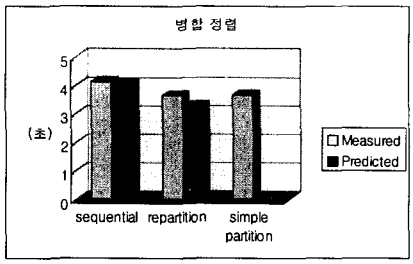

그림 4 병합 정렬 실행 결과

을 실제 실행하여 소요시간을 측정하는 것은 실행된 일부 반복 구간이 특이하게 짧거나 길지 않는 한, 재분할에 필요한 의미있는 정보를 제공한다고 볼 수 있다. 실제로 단순 분할, 재분할, 단일 노드 처리 시간 모두에 대해 실행시간을 20~30% 일률적으로 짧게 예측함으로써 결과적으로 수행 시간을 단축시킨 것이 이를 증명한다.

병합 정렬은 n개의 자료가 주어졌을 때 이를 일정 구간으로 나누고 구간 내부에서 정렬을 수행한 뒤 정렬된 구간들을 합병하는 방식으로 수행되는 정렬이다. 여기서는 100만개의 자료를 몇 개의 부분 집합으로 재구성하고 이들 각각을 병합 정렬하고 최종적으로 이들을 합병하도록 C 코드를 작성하여, 구현된 컴파일러로 병렬화하였다. 재분할된 코드는 단일 노드에 비해 20%의 수행시간을 단축하였고 단순 분할된 코드에 비해서는 1%의 수행시간을 단축하였다. 단순 분할된 코드에 비해 재분할된 코드가 성능 개선 효과가 두드러지지 않은 것은 통신량과 처리 능력이 균형을 이루어 최적 분할과 균등 분할이 거의 동일하기 때문이다.

5. 결 론

병렬처리 시스템들은 종종 병렬성에 대한 지나친 집착으로 의미없는 병렬화를 수행하고 성능 저하를 겪곤 한다. 프로그램의 일부를 다른 처리 노드에 나누어 실행시킬 것인가 말 것인가의 결정은 그 분할의 실행시간과 병렬화에 수반되는 통신 부담 정보에 근거하여 수행되어야 한다.

일반적으로 특정 분할의 정확한 실행시간을 예측하거나 통신 시간을 예측하기란 매우 어려우므로, 병렬처리를 위해 프로그램을 어떻게 분할할 것인가 결정하는 문제도 쉽지 않다. 그러나 우리가 어떻게 최고의 성능 향상을 얻을 것인가에 집착하지 않고, 프로그램의 실행시간을 어떻게 줄일 수 있을까의 문제에 집중한다면 테스트 프로그램을 수행시켜서 얻은 예측치를 이용하여 실행시간과 부담에 관련된 정보를 대략적으로나마 알 수 있을 것이다.

본 연구에서는 이러한 문제 의식과 발상에 기초하여 이기종 연결형 병렬처리 시스템을 구축하고, 이를 기반으로 통신 부담 정보와 실행 성능 정보를 추출하여 병렬화 결정을 평가할 뿐 아니라 병렬 루프의 최적 분할을 찾아주는 컴파일러를 개발하였다. 본 연구를 통해 전문적 병렬 프로그래밍 능력을 갖지 못한 일반 프로그래머도 자신의 응용 분야에서 기존의 C 언어로 프로그래밍하면 저가의 워크스테이션 네트워크를 활용하는 병렬 프로그램을 얻을 수 있게 되었다.

설계, 구현된 시스템은 300MHz 주파수로 동작하는 AlphaPC 2대와 333MHz로 동작하는 Pentium-II PC 2대를, TCP/IP 프로토콜을 이용하는 100 Mbps 이더넷으로 연결된 시스템 상에서 실험되었다. 실험은 루프의 길이가 매우 짧은 병렬 루프(daxpy)와 조금 더 긴 병합 정렬, 3중 루프라서 좀더 실행시간이 더 걸리는 행렬 곱셈, 루프 몸체의 길이가 대단히 긴 순회 세일즈맨 문제, 벤치마크 슈트인 NAS 등을 통해 이루어졌다. 실제 프로그램과 커널 모음인 NAS 벤치마크에 대해서 구현된 컴파일러 시스템은 의미있는 병렬화를 해내지 못했으며, NAS 벤치마크에서 가장 흔하게 찾아볼 수 있는 의미없는 병렬화인 daxpy 코드의 경우 그냥 병렬화할 경우 80배의 시간을 요구하지만 본 컴파일러의 분석기는 이를 포착하여 의미없는 병렬화를 생략시킴으로써 성능상 지연을 막는 능력을 보였다. 본 연구의 근본 취지 자체가 의미 없는 병렬화로 인한 성능 저하를 예측 발견하여 의미 없는 과도한 병렬화를 회피하는 것이었던 만큼, 본래의 목적을 달성하였다고 생각된다.

그 외의 벤치마크 프로그램들에 대해서는 10%에서 40%까지 성능을 단축시킴으로써 이기종 컴퓨터들의 네트워크가 컴파일러에 의해 자동적으로 병렬화된 병렬 프로그램의 실행 컴퓨터로서 의미가 있음을 보였다. 다른 측면으로는 이기종 컴퓨터 네트워크를 대상으로 하는 자동적 병렬화 컴파일러가 생성하는 병렬 프로그램이 단일 노드 순차 실행 프로그램에 비해 성능 개선을 기대할 수 있음도 보였다.

비록 4대의 컴퓨터에 의한 수행시간 단축 효과가 10~40%로 상식적, 일반적 병렬처리 컴퓨터에 비해 떨어지지만, 이것은 특화된 전용의 고속 상호 연결망을 채용하지 않은 것에 주된 이유가 있고 한편으로는 시스템 구성에 이용된 AlphaPC가 마스터 노드 Pentium-II에 비해 사실상 절반 가량의 성능밖에 낼 수 없는 데도 주요한 이유가 있다. 성능이 차이가 나는 노드로 구성되는

시스템에 대해 시스템 구성 대수 증가에 따른 선형적 성능 향상을 기대하는 것은 무리이다. 이는 본 연구의 기본 취지가 시스템 노후화로 인해 사장되는 컴퓨팅 파워를 병렬처리에 투입하여 성능상 개선과 자원 재활용을 도모하자는 근본 취지에도 부합한다 하겠다.

본 연구는 유사한 시도가 알려져 있지 않은 만큼, 이후 유사 연구의 기저가 될 수 있다는 점에서도 의미가 있다 하겠다. 그러나, 제안된 시스템은 공개된 SUIF 컴파일러에서 찾아주는 제한된 병렬성만을 이용할 뿐만 아니라 최적화되지 않은 통신 패러다임을 이용하는 등의 여러 문제점들을 가지고 있다. 향후에는 지속적인 문제점 해결과 시스템 안정화를 통해 자원 재활용을 통한 고성능 계산 능력의 확보를 목표로 계속 추구할 것이다.

참 고 문 헌

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson, "A Case for Networks of Workstations: NOW," *IEEE Micro*, Feb, 1995.

[2] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Communications of the ACM*, Vol. 35, No. 8 (Aug. 1992), Pages 66-80

[3] High Performance Fortran Language Specification Version 1.0, May 1993.

[4] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E Bugnion Bugnion and M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF compiler," *IEEE Computer*, December 1996.

[5] D. A. Padua et al., "Polaris: A new-generation parallelizing comiler for MPPs," *Technical Report CSRD-1306*, Center for Supercomputing Research and Development, Univ. of Illinois at Uurbana-Champaign, June 1993.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skellum, "A high-performance, portable implementation of the MPI message-passing interface standard," *Parallel Computing*, 22:789-828, 1996

[7] MPI Primer / Developing With LAM (manual), Ohio Supercomputer Center, 1996.

[8] A.V.Aho et al., *Compilers - Principles, Techniques, and Tools*, Addison Wesley, pp.585-711, 1986.

[9] Keqin Li, "Predicting the Performance of Partitionable Multiprocessors," *Proc. Of PDPTA96 International Conference*, pp 1350-1353., 1996.

[10] Edward W. Felton et al., "Early Experience with Message-Passing on the SHRIMP Multiprocessor," Proc. of ISCA'96, pp. 296-307

[11] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transaction on Computers, Vol C-30, No 7, pp. 478~490, July 1981.

[12] Anant Agarwal et al., "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors," IEEE Transactions on Parallel and Distributed Systems Vol. 6, No. 9, September, 1995.

[13] Rudolf Eigenmann, Insung Park, and Michael J. Voss. "Are Parallel Workstations the Right Target for Parallelizing Compilers?," *Lecture Notes in Computer Science, No. 1239: Languages and Compilers for Parallel Computing*, March 1997.

권 대 석

1992년 서울대학교 계산통계학과 이학사. 1994년 서울대학교 대학원 계산통계학과 전산과학 전공 이학석사. 1994년 ~ 현대 서울대학교 대학원 전산과학과 박사과정. 관심분야는 병렬처리 구조, 병렬화 컴파일러, 인공지능, 인지과학

김 홍 환

정보과학회논문지 : 시스템 및 이론 제 27 권 제 1 호 참조

한 상 영

정보과학회논문지 : 시스템 및 이론 제 27 권 제 1 호 참조