Linux Parallel Processing HOWTO

Hank Dietz, pplinux@ecn.purdue.edu v980105, 5 January 1998

1장-2장 이호, (i@flyduck.com), 3장 이후 선정필, (simje@maninet.com) 1999년 12월 3일, 최종 업데이트: 2000년 4월 18일

병렬처리(Parallel Processing)는 프로그램을 동시에 실행할 수 있는 여러 조각으로 나누어 각자 자신의 프로세서에서 실행함으로써 프로그램 수행 속도를 빠르게 한다는 개념이다. 프로그램을 *N*개의 프로세서에서 실행하면 하나의 프로세서 실행하는 것보다 *N*배까지 빨라질 수 있다. 이 문서는 리눅스 사용자들이 사용할 수 있는 네가지 병렬 처리에 대한 접근법을 다룬다. SMP 리눅스 시스템, 네트웍으로 연결된 리눅스 시스템의 클러스터, 멀티미디어 명령어(MMX같은)를 이용한 병렬 수행, 하나의 리눅스 시스템이 호스트하는 부속 (병렬) 프로세서(attached processor)가 이들이다.

목 차

제 1 절 소개

병렬처리(Parallel Processing)는 프로그램을 동시에 실행할 수 있는 여러 조각으로 나누어 각자 자신의 프로세서에서 실행함으로써 프로그램 수행 속도를 빠르게 한다는 개념이다. 프로그램을 N개의 프로세서에서 실행하면 하나의 프로세서 실행하는 것보다 N배까지 빨라질 수 있다.

오랫동안 특별히 디자인한 "병렬 컴퓨터(parellel computer)"에서 여러개의 프로세서를 사용할 수 있었다. 이런 경향에 따라 리눅스는 현재 하나의 컴퓨터 내에서 여러개의 프로세서가 같은 메모리와 버스 인터페이스를 공유하는 **SMP** 시스템(종종 "서버"로 팔리는)을 지원한다. 이 외에도 여러대의 컴퓨터를 그룹을 지어 (예를 들어 각각 리눅스를 실행하고 있는 PC들의 그룹) 네트웍으로 서로 연결하여 병렬처리 **클러스터**(parellel-processing cluster)를 만들 수 있다. 리눅스를 이용한 병렬 컴퓨팅의 세번째 방법은 **멀티미디어 확장 명령어**(multimedia instruction extensions, MMX)를 사용하여 숫자 데이터 벡터를 병렬로 처리하는 것이다. 마지막으로 리눅스 시스템을 전용으로 **부속** 병렬처리 엔진(attached parellel processing compute engine)의 "호스트"로 사용하는 것도 가능하다. 이 문서에서는 이 모든 접근방법들을 자세히 다루도록 하겠다.

1.1 병렬처리가 내가 바라던 것인가?

여러개의 프로세서를 사용하는 것은 많은 연산의 처리하는 속도를 빠르게 할 수 있지만, 대부분의 응용프로그램들은 병렬처리라고 해서 아직 나아지는게 없다. 기본적으로 병렬처리는 다음 경우에 해당할 때 적당하다 :

- 응용 프로그램이 여러개의 프로세서를 효과적으로 사용할 수 있도록 병행성을 가지고 있어야 한다. 어느정도 이는 프로그램 중에서 각기 다른 프로세서에서 독립적으로 동시에 실행할 수 있는 부분들을 파악하는 문제이다. 특정 시스템을 사용하여 병렬로 실행하는 경우, 어떤 것들은 병렬로 실행하는게 실제로 더 느린 경우가 있을 수 있다. 예를 들어 하나의 컴퓨터에서 4초가 걸리는 프로그램이 네 대의 컴퓨터에서 각각 1초만에 실행을 끝낸다 하더라도, 이들 컴퓨터가 서로의 동작을 통합하는데 3초나 그 이상의 시간이 걸린다면 아무런 속도개선이 이루어지지 않는다.
- 관심을 가지고 있는 특정 응용프로그램이 이미 병렬화(병렬처리의 이점을 활용하여 다시 작성된) 되었거나, 병렬처리의 이점을 활용하는 최소한의 새로운 코딩을 하려고 해야 한다.
- 연구분야에 관심있거나 어느정도 익숙한 사람이 병렬처리를 포함하도록 나서야 한다. 리눅스 시스템을 이용한 병렬처리가 반드시 어려운 것은 않지만, 대부분의 컴퓨터 사용자에겐 친숙하지 않고, "아무것도 모르는 사람들을 위한 병렬처리"같은 책도 아직 없는 상황이다. 이 HOWTO 문서가 알아야 할 모든것은 가지고 있진 않더라도 좋은 출발점이 될 것이다.

좋은 소식은 위의 내용이 모두 해당한다면, 복잡한 계산을 수행하거나 방대한 데이터를 처리하는 프로그램의 경우, 리눅스를 이용한 병렬처리가 슈퍼컴퓨터급의 성능을 발휘할 수 있다는 것이다. 더군다나 그것도 당신이 이미 가지고 있을 값싼 하드웨어를 사용하여 할 수 있다. 보너스로 병렬 리눅스 시스템이 바쁘게 병렬 작업을 수행하고 있지 않을 때는 다른 용도로 쉽게 사용할 수 있다.

병렬처리가 당신이 바라던 것이 아니더라도 어느정도 소소한 성능향상을 바란다면, 여전히 할 수 있는 일이 몇가지 있다. 예를 들어, 순차처리를 하는 프로그램들은 빠른 프로세서를 사용하고, 메모리를 추가하고, IDE 디스크를 빠른 와이드 SCSI 디스크로 바꾸는 등의 방법을 사용할 수 있다. 당신이 관심을 가지는 것이 이거라면 바로 6.2 (성능 문제)장으로 넘어가고, 그렇지 않으면 계속 읽어주기 바란다.

병렬 처리가 여러분이 원하는 것이 아니더라도 여러분이 적어도 가장 온건한 성능 개선을 하고자 한다면 여러분이 할 수 있는 것들이 아직 남아 있다. 예를 들어서 여러분은 좀 더 빠른 프로세서, 메모리 추가, IDE 디스크를 빠른 와이드 SCSI로 바꾸는 등의 일을 함으로써 시퀀셜 프로그램들의 성능을 개선할 수 있다. 이

것이 여러분이 관심이 있는 모든 것이라면 섹션 6.2 (성능에 대한 논란)로 점프하라; 그렇지 않다면 계속 읽기 바란다.

1.2 용어

여러 해 동안 많은 시스템에서 병렬처리를 사용해왔지만, 대부분의 컴퓨터 사용자들은 여전히 좀 낯설 것이다. 따라서 병렬처리의 여러 방법들을 살펴보기 전에, 몇가지 일반적으로 사용하는 용어들에 익숙해지는 것이 필요하다.

SIMD (SMingle Instruction stream, Multiple Data stream, 단일 명령어, 다중 데이터 스트림) :

SIMD는 모든 프로세서가 똑같은 연산을 동시에 실행하지만, 각 프로세서가 자신만의 데이터에 대해 연산을 수행할 수 있는 병렬 실행 모델을 가리킨다. 이 모델은 배열의 모든 원소에 대해서 똑같은 연산을 수행하는 개념에 자연히 들어맞으며, 따라서 종종 벡터나 배열 처리와 관련된다. 모든 연산이 본래 동기화되어있으므로, SIMD 프로세서간의 상호작용은 대체로 쉽고 효과적으로 구현할 수 있다.

MIMD (Multiple Instruction stream, Multiple Data stream, 다중 명령어, 다중 데이터 스트림) :

MIMD는 각 프로세서가 근본적으로 독립적으로 동작하는 병렬 실행 모델을 가리킨다. 이 모델은 프로그램을 기능적인 토대에 바탕하여 병렬 실행할 수 있는 것으로 쪼개는 개념에 대부분 자연스럽게 들어맞는다. 예를 들어, 한 프로세서는 새로운 엔트리를 그래픽 화면으로 만들고 있을 때, 다른 프로세서는 데이터베이스 파일을 갱신하는 것이다. 이는 SIMD 보다는 더 유연한 모델이지만, 한 프로세서의 연산과 다른 프로세서의 연산의 상대순위가 바뀌는 시간 변화로 인하여 프로그램이 실패할 수 있는 경주 상황(race conditions)라는 악몽의 디버깅을 감수해야 한다.

SPMD (Single Program, Multiple Data, 단일 프로그램, 다중 데이터) :

SPMD는 MIMD의 제한된 버전으로 모든 프로세서가 같은 프로그램을 실행하는 것이다. SIMD와는 달리, SPMD 코드를 실행하는 각 프로세서는 프로그램을 실행 과정에서 다른 제어 흐름 과정을 따를 수 있다.

통신 대역폭 (Communication Bandwidth) :

통신 시스템의 대역폭은 데이터 전송을 시작한 때부터 어떤 단위의 시간동안 전송할 수 있는 데이터의 최대크기이다. 직렬 연결에서는 대역폭을 대개 baud 또는 비트/초 (b/s)로 표시하는데, 일반적으로 이것의 1/10에서 1/8이 바이트/초 (B/s)에 해당한다. 예를 들어, 1200 baud 모뎀은 약 120 B/s의 속도로 전송을 하고, 반면에 155 Mb/s ATM 네트웍 연결은 이보다 130000배 가량 빠른, 약 17 MB/s의 속도로 전송을 한다. 큰 대역폭은 프로세서 사이에 큰 데이터 블럭을 효율적으로 전송할 수 있게 한다.

통신 지체 (Communication Latency) :

통신 시스템의 지체(latency)는 보내고 받는 소프트웨어의 오버헤드를 포함하여, 한 객체를 전송하는데 걸리는 최소한의 시간을 말한다. 지체는 병렬처리에서 매우 중요한데, 병렬 실행으로 속도를 향상시킬 수 있는 코드 조각의 최소 실행 시간인, 최소 유용 알갱이 크기(minimum useful grain size)를 결정하기 때문이다. 기본적으로 코드 조각을 실행하는 시간이 결과값을 전송하는 시간(즉, 지체)보다 짧을 때, 그 코드 조각을 결과값을 필요로 하는 프로세서에서 직렬로 실행하는 것이 병렬로 실행하는 것보다 더 빠르다. 직렬로 실행하는 것은 통신 오버헤드가 없기 때문이다.

메시지 전달 (Message Passing) :

메시지 전달은 병렬 시스템 내부에서 프로세서간의 상호작용을 위한 모델이다. 일반적으로, 메시지는 한 프로세서에 있는 소프트웨어에서 만들어지고, 상호연결 네트웍을 통하여 다른 프로세서로 전달되어, 여기서 이를 받아 메시지 내용에 따라 동작하게 된다. 각 메시지를 처리하는 오버헤드(지체)가 클 수 있지만, 대개 각 메시지가 어느 정도 크기의 정보를 가질 수 있는지에는 거의 제한을 두지 않는다. 그래서 메시지 전달은 큰 대역폭을 초래하기도 하며, 한 프로세서에서 다른 프로세서로 큰 데이터 블

력을 전달하는 것을 매우 효율적인 방법으로 처리도록 되어 있다. 그렇지만, 값비싼 메시지 전달 연산의 필요를 최소화할 수 있도록, 병렬 프로그램에 있는 자료구조는 프로세서 간에 널리 퍼져 있어서 각 프로세서가 참조하는 대부분의 데이터는 자신의 지역 메모리 상에 있도록 해야 한다. 이러한 작업을 데이터 배치(data layout)라고 한다.

공유 메모리 (Shared Memory) :

공유 메모리는 병렬 시스템 내부에서 프로세서간의 상호작용을 위한 모델이다. 리눅스를 실행하고 있는 멀티프로세서 펜티엄 컴퓨터같은 시스템은 물리적으로 프로세서간에 하나의 단일 메모리를 공유한다. 따라서 한 프로세서가 공유 메모리에 값을 기록하면, 다른 어떤 프로세서든지 이 값을 직접 읽을 수 있다. 이와 달리 논리적인 공유 메모리는 각 프로세서가 자신만의 메모리를 가지며, 지역 메모리에 없는 메모리를 참조하면 이를 해당하는 프로세서간 통신으로 변환해줌으로써 구현한다. 이들 각각의 공유 메모리 구현은 일반적으로 메시지 전달보다 사용하기 쉽게 되어 있다. 물리적인 메모리 공유는 큰 대역폭을 가지며 지체가 적지만, 이는 단지 여러 프로세서가 동시에 버스에 접근하려하지 않을 때만이다. 따라서 데이터 배치(data layout)는 여전히 성능에 큰 영향을 미칠 수 있으며, 캐시 효과 등은 어떻게 배치하는 것이 가장 좋은 것인지 결정하기 힘들게 만든다.

집합 함수 (Aggregate Functions) :

메시지 전달과 공유 메모리 모델에서 통신은 모두 하나의 단일 프로세서에서 시작한다. 이와 반대로 집합 함수 통신은 본래 모든 프로세서 그룹이 서로 작용할 수 있는 병렬 통신 모델이다. 이런 작용의 가장 간단한 것은 장벽 동기화(barrier synchronization)로, 개별 프로세서들이 그룹에 있는 모든 프로세서가 장벽에 도달하길 기다리는 것이다. 개별 프로세서가 장벽에 도착하면서 부수효과(side effect)로 데이터를 출력하면, 통신 하드웨어는 모든 프로세서에서 수집한 값들에 임의의 함수를 적용한 결과값을 각 프로세서에게 전달할 수 있다. 예를 들어, 그 결과값은 "어떤 프로세서가 해를 찾았느냐"는 질문의 대답일 수도, 각 프로세서에서 온 값들의 합일 수도 있다. 지체(latency)는 매우 적겠지만, 하나의 프로세서가 차지하는 대역폭 역시 적은 경향이 있다. 전통적으로 이 모델은 데이터 값을 분산하기보다는 병렬 실행을 제어하는데 주로 사용된다.

총괄 통신 (Collective Communication) :

이는 집합 함수(aggregate function)의 다른 이름으르, 대부분 다중 메시지 전달 연산을 이용하여 구축된 집합 함수를 가리키는데 사용된다.

SMP (Symmetric Multi-Processor, 대칭형 멀티프로세서)

SMP는 일련의 프로세서들이 서로 대등하게 함께 동작하여, 어떤 작업 조각이든지 어떤 프로세서에서든 똑같이 실행될 수 있는 운영체제 개념을 말한다. 대체로 SMP는 MIMD와 공유메모리를 결합한 것이다. IA32 계열에서 SMP는 일반적으로 MPS(Intel Multi-Processor Specification, 인텔 멀티프로세서 규약)와 호환된다는 것을 의미한다. 앞으로는 이것은 "Slot 2"를 의미하게 될 것이다...

SWAR (SIMD Within A Register, 레지스터에서의 SIMD) :

SWAR는 하나의 레지스터를 여러개의 정수 항목으로 쪼개고 레지스터 너비의 연산을 사용하여 이들 항목들에 SIMD 병렬 계산을 수행한다는 개념을 가리키는 일반적인 용어이다. k-bit 레지스터와 데이터 통로, 함수 단위를 갖는 기계가 있을 때, 오래전부터 보통의 레지스터 연산을 사용하여 n개의 k/n 비트 항목 값에 SIMD 병렬 연산을 할 수 있다고 알려져왔다. 이런 방식의 병렬성은 보통의 정수 레지스터를 사용하여 구현할 수 있지만, 많은 고성능 마이크로프로세서들은 멀티미디어 위주 작업에 이 기법의 성능을 높이기 위해 최근 특별 명령어들을 추가했다. 인텔/AMD/Cyrix의 MMX(MultiMedia eXtension)를 비롯하여, 디지털(Digital) Alpha의 MAX(MultimediA eXtensions), 휴렛- 팩커드(Hewlett-Packard) PA-RISC의 MAX(Multimedia Acceleration eXtensions), MIPS의 MDMX(Digital Media eXtension, "Mad Max"라고 발음한다), 선(Sun) SPARC의 V9 VIS(Visual Instruction Set) 등이 있다. MMX에 동의한 세 회사를 제외하고, 이들 확장 명령어들은 대충은 비슷하지만, 서로 호환되지는 않는다.

부속 프로세서 (Attached Processors) :

부속 프로세서는 본질적으로 특별한 유형의 계산 속도를 가속하기 위한 호스트 시스템에 연결된 특별한 목적을 가진 컴퓨터이다. 예를 들어, PC에 있는 많은 비디오와 오디오 카드는 제각기 일반 그래픽 연산과 오디오 DSP(Digital Signal Processing, 디지털 신호 처리) 속도를 높이도록 디자인된 부속 프로세서를 가지고 있다. 또한 배열에 대한 산술 연산 속도를 빠르게 하기 위한, 넓은 범위의 부속 배열 프로세서(attached array processor)들이 있다. 많은 상업용 슈퍼컴퓨터들은 실제로 워드스테이션 호스트와 부속 프로세서로 되어 있다.

RAID (Redundant Array of Inexpensive Disk, 여분의 값싼 디스크 배열) :

RAID는 디스크 I/O의 신뢰성과 대역폭을 늘리는 간단한 기술이다. 여기에는 여러가지 서로 다른 변형이 있지만, 모두 두가지 핵심 개념을 공유하고 있다. 먼저, 각 데이터 블럭은 n+k 디스크 드라이브 그룹으로 줄을 지어, 각 드라이브는 단지 데이터의 1/n 만큼 읽고 쓰기만 하지만, 각 드라이브 대역폭의 n배의 대역폭을 가지게 된다. 두번째로, 여분으로 데이터를 기록하여, 한 디스크 드라이브가 실패하더라도 데이터를 복구할 수 있도록 한다. 이것은 매우 중요한데, 그렇지 하지 않으면 n+k 드라이브 중 하나가 실패한 경우 전체 파일 시스템이 날라갈 수 있기 때문이다. `http://www.dpt.com/uraiddoc.html`에 가면 RAID 전반에 관한 좋은 개요가 있다. 리눅스 시스템에서의 RAID 옵션에 대한 정보는 `http://linas.org/linux/raid.html`에서 찾을 수 있다. 전문 RAID 하드웨어 지원과는 별도로, 리눅스는 하나의 리눅스 시스템이 여러개의 디스크를 호스트하는 소프트웨어 RAID 0, 1, 4, 5도 지원한다. 자세한 것은 소프트웨어 RAID mini-HOWTO와 다중 디스크 튜닝(Multi-Disk Tuning) mini-HOWTO를 참조하기 바란다. 클러스터에 있는 여러 기계에 있는 디스크 드라이브들의 RAID는 직접적으로 지원되지 않는다.

IA32 (Intel Architecture, 32-bit, 인텔 32비트 아키텍쳐) :

IA32는 실제로 병렬처리하고는 관련이 없고, 단지 일반적으로 인텔 386 프로세서와 호환된는 명령어 집합을 가지는 프로세서들의 부류를 가리킨다. 기본적으로, 286 다음에 나온 모든 인텔 x86 프로세서는 IA32의 특징인 32비트 플랫 메모리 모델(flat memory model)과 호환된다. AMD와 Cyrix 역시 수많은 IA32 호환 프로세서를 만든다. 리눅스가 주로 IA32 프로세서에서 발전해왔으며, IA32가 상품시장의 중심에 있기 때문에, PowerPC나 Alpha, PA-RISC, MIPS, SPARC 등의 다른 프로세서와 구별하여 IA32라는 용어를 사용하는 것이 편리하다. 곧 출시될 IA64(EPIC, Explicitly Parallel Instruction Computing, 명시된 병렬 명령 계산을 지원하는 64비트 프로세서)는 아마도 복잡한 문제가 되겠지만, 처음 나오게 될 IA64 프로세서인 머세드(Merced)는 1999년까지는 제품이 나오진 않을 예정이다.

COTS (Commercial Off-The-Shelf, 상업용 기성품)

많은 병렬 슈퍼컴퓨터 회사들이 사라지면서, COTS는 병렬 계산 시스템의 필요조건으로 일반적으로 다루어지게 되었다. 아주 이론적으로 하면, PC를 사용하는 유일한 COTS 병렬처리 기법은 SMP Windows NT 서버와 여러 MMX Windows 응용프로그램같은 걸로 만들어진 것이다. COTS 개념의 기반은 사실상 개발 시간과 비용의 최소화이다. 따라서 더 유용하고, 더 일반적인, COTS의 의미는 적어도 대부분의 서브시스템은 기성 제품 시장에서 이득을 얻어야 하지만, 다른 기술들은 효율적으로 사용될 수 있는 곳에 사용해야 한다는 것이다. 대부분의 경우, COTS 병렬처리는 노드는 기성 PC이지만 네트웍 인터페이스와 소프트웨어는 어느정도 맞춤으로 만든 클러스터를 가리킨다. 대개 실행할 리눅스와 응용프로그램 코드는 자유롭게 구할 수 있지만 (copyleft이거나 public domain인), 문자 그대로 COTS는 아니다.

1.3 예제 알고리즘

이 HOWTO에서 언급하고 있는 여러가지 병렬 프로그래밍 접근 방법들의 사용법을 좀 더 잘 이해할 수 있도록, 예제 문제를 하나 다루어보도록 하자. 비록 간단한 병렬 알고리즘이지만, 여러 다른 병렬 프로그래밍 시스템을 시연하는데 사용해왔던 알고리즘을 선택함으로써, 각 접근방법을 비교하고 대조하는 것이 조금 더 쉬울 것이다. M.J.Quinn의 책 (Parallel Computing Theory And Prictice (병렬 계산 이론과 실습)); 2판, McGraw Hill, New York, 1994에서는, 다양한 서로 다른 병렬 슈퍼컴퓨터 프로그래밍 환경(예를 들어, nCUBE 메시지 전달, 순차 공유 메모리(sequent shared memory))을 시연하기 위해, Pi 값을 계산하는 병렬 알고리즘을 사용하고 있다. 이 HOWTO에서, 우리도 똑같은 기본 알고리즘을 사용하도록 하자.

이 알고리즘은 x의 정사각형 아래에 있는 영역을 합하여 Pi의 근사값을 계산한다. 순수한 순차 C 프로그램으로 만든다면 알고리즘은 다음과 비슷할 것이다.

```
#include <stdlib.h>;
#include <stdio.h>;

main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the computation */
  sum = 0;
  for (i=0; i<intervals; ++i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;

  printf("Estimation of pi is %f\n", sum);

  return(0);
}
```

그렇지만 이 순차 알고리즘은 쉽게 "곤란한 병렬(embarrassingly parallel)" 구현이 된다. 이 영역들은 간격(intarval)별로 쪼개고, 프로세서가 몇개라도 프로세서간에 상호작용할 필요 없이, 자기에게 할당된 간격을 독립적으로 합할 수 있다. 일단 지역별로 합이 계산되었다면, 전체합을 만들기 위해 서로 더해야 한다. 이 과정은 프로세서간에 어느정도 레벨의 조정과 통신을 필요로 한다. 마지막으로 전체 합은 Pi값의 근사치가 되어 한 프로세서에서 이를 출력하게 된다.

이 HOWTO에서는, 이 알고리즘의 여러가지 병렬 구현이 나오며, 각각은 다른 프로그래밍 방법을 사용한다.

1.4 이 문서의 구성

이 문서의 나머지는 다섯개 부분으로 나뉘어져 있다. 2, 3, 4, 5장은 리눅스를 이용한 병렬처리를 지원하는 세가지 다른 유형의 하드웨어 구성을 다루고 있다.

- 2장은 SMP 리눅스 시스템을 다룬다. 이는 공유 메모리를 이용한 MIMD 실행을 직접적으로 지원하며, 메시지 전달 역시 쉽게 구현된다. 리눅스는 16개의 프로세서를 갖는 SMP 구성까지 지원하지만, 대부분의 SMP PC 시스템은 두개나 네개의 똑같은 프로세서를 가지고 만들어진다.
- 3장은 각각 리눅스를 실행하고 있는 기계들을 네트웍으로 연결한 클러스터를 다룬다. 클러스터는 MIMD 실행과 메시지 전달, 그리고 대개 논리적 공유 메모리를 직접 지원하는 병렬처리 시스템으로 사용할 수 있다. 사용한 네트웍 방법에 따라 SMP 실행을 흉내내고, 집합 함수(aggregate function) 통신도 지원할 수 있다. 클러스터로 연결된 프로세서의 숫자는 두개에서 수천개까지 될 수 있는데, 이 숫자는 주로 네트웍을 구성하는 물리적인 배선에 의해 제한을 받는다. 어떤 경우, 클러스터에 서로 다

른 유형의 기계들을 혼합할 수도 있다. 예를 들어, DEC Alpha와 펜티엄 리눅스 시스템을 결합할 수 있는데, 이런 것을 가리켜 **이질 클러스터**(heterogeneous cluster)라고 한다.

- 4장에서는 SWAR, 즉 레지스터에서의 SIMD(SIMD Within A Register)를 다룬다. 이것은 매우 제한적인 유형의 병렬 실행 모델이지만, 반면에 일반적인 프로세서에 이미 구현되어 있는 기능이기도 하다. 최근에 근래의 마이크로프로세서에 MMX (그리고 다른 것들도) 확장 명령어들이 추가되면서 이런 접근방법이 더 효율적이 되었다.
- 5장에서는 리눅스 PC를 간단한 병렬처리 시스템의 호스트로 사용하는 것을 다룬다. 꼽는 카드나 외부의 박스 형태로, 부속 프로세서는 리눅스 시스템에게 특정 종류의 응용프로그램에 대한 엄청난 처리 능력을 줄 수 있다. 예를 들어, 여러개의 DSP 프로세서를 제공하는 값싼 ISA카드를 이용하여, 경계계산 문제(compute-bound problem)를 위한 수백 MFLOPS의 처리 능력을 가질 수 있다. 그렇지만 이들 추가되는 보드들은 단지 프로세서일 뿐이다. 이들은 일반적으로 OS를 실행하거나 디스크나 콘솔 I/O 능력 등을 가지고 있지 않다. 이런 시스템을 유용하게 사용하기 위해 리눅스 "호스트"가 이들 기능들을 제공해야 한다.

이 문서의 마지막 장은 위에서 다룬 접근 방법들에 속하지 않는, 리눅스를 이용한 병렬처리에서 일반적으로 가지고 있는 관심들을 다룬다.

이 문서를 읽을 때 아직 우리가 모든 것들을 다 테스트해보진 못했다는 것과 여기서 다루는 내용의 많은 부분은 "아직 연구중인 특성"("생각했던 것처럼 잘 동작하지 않는다"는 것을 더 좋게 표현한 말이다 :-)이라는 것을 명심하기 바란다. 그렇지만 리눅스를 이용한 병렬처리는 현재 유용하며, 점점 더 많은 그룹들이 이를 더 잘 사용하기 위해 작업을 진행중이다.

이 HOWTO 문서를 작성한 사람은 Hank Dietz 박사로 현재는 West Lafayette 47907-1285에 있는 Purdue 대학의 전기 및 컴퓨터 공학(Electrical and Computer Engineering)의 부교수(Associate Professor)이다. Dietz는 리눅스 문서화 프로젝트(Linux Documentation Project, LDP)의 지침에 따라 이 문서에 대한 권한을 갖는다. 이 문안을 정확하고 공정하게 만들기 위해서 많은 노력을 했지만, Dietz나 Purdue 대학 모두 어떠한 문제나 에러에 대한 책임이 없으며, Purdue 대학은 여기서 다룬 어떠한 작업이나 결과물도 보증하지 않는다.

제 2 절 SMP 리눅스

이 문서는 병렬처리를 위해 *SMP 리눅스* `<http://www.uk.linux.org/SMP/title.html>` 시스템을 어떻게 사용할 수 있는지에 관해 간단하게 개요를 제시한다. SMP 리눅스에 대한 가장 최근 정보는 아마도 SMP 리눅스 프로젝트의 메일링 리스트에서 얻을 수 있을 것이다. 이 리스트에 가입하려면 편지 본문에 `subscribe linux-smp` 라고 적어 `majordomo@vger.rutgers.edu`로 편지를 보내면 된다.

SMP 리눅스가 정말 제대로 동작하는가? 1996년 6월, 나는 새로운 상표의 (사실은 한물 간 품종이었지만 새 상표였다 ;-) 두개의 100MHz 펜티엄 프로세서를 가지는 시스템을 구입했다. 조립을 마친 시스템은 두 개의 프로세서와 Asus 마더보더(motherboard), 256K 캐시, 32M RAM, 1.66G 하드디스크, 6배속 CDROM, Stealth 64 그래픽 카드와 15인치 모니터로, 이를 마련하는데 모두 1800$가 들었다. 이 가격은 이와 비슷한 사양의 프로세서 하나인 시스템보다 단지 몇백 달러정도 비싼 거였다. 제대로 동작하는 SMP 리눅스를 구할려면, 그저 보통의 단일 프로세서 리눅스를 설치하고, makefile에서 `SMP=1`을 막고 있는 주석을 해제하여 (비록 `SMP`를 1로 설정하는 것이 조금은 반어적이라는 것을 알지만) 커널을 다시 컴파일하고, `lilo`에게 새로운 커널을 알려주기만 하면 된다. 이 시스템은 매우 잘 동작하였고, 안정적이기도 하여, 지금까지 사용해 온 나의 주 워크스테이션의 역할을 수행하기에 충분했다. 요약하면, SMP 리눅스는 정말로 제대로 동작한다.

다음 질문은 SMP 리눅스가 공유 메모리를 사용하는 병렬 프로그램을 작성하고 실행하는 데 있어 얼만큼이나 고수준으로 지원을 해주느냐이다. 1996년 초에는 이런 것은 별로 많지 않았다. 그러나 이제 많은 것이 변했다. 예를 들어, 이제는 매우 완벽한 POSIX 쓰레드(thread) 라이브러리가 있다.

공유 메모리 방식을 사용하는 것보다는 성능이 떨어질 수도 있지만, SMP 리눅스 시스템에서는 원래 소켓 통신을 사용하는 워크스테이션 클러스터(cluster)에서 동작하도록 개발된 대부분의 병렬 처리 소프트웨어

도 사용할 수 있다. 소켓(3.3 장을 보라) 방식은 SMP 리눅스 시스템 뿐만 아니라, 여러개의 SMP 시스템을 네트웍으로 연결한 클러스터에서도 사용할 수 있다. 그렇지만 소켓은 SMP에서는 상당한 양의 불필요한 오버헤드(overhead)를 가지게 된다. 이런 오버헤드의 대부분은 커널 즉 인터럽트 핸들러에서 일어난다. SMP 리눅스에서는 보통 동시에 하나의 프로세서만이 커널 모드에 있을 수 있고, 부트 프로세서만이 인터럽트를 처리할 수 있도록 인터럽트 컨트롤러가 설정되어 있기 때문에 문제는 더 심각해진다. 그럼에도 불구하고, 전형적인 SMP 통신 하드웨어가 클러스터 네트웍보다는 훨씬 좋기 때문에, 원래 클러스터에서 사용하려고 만든 클러스터용 소프트웨어도 SMP에서 더 좋은 성능을 보인다.

이 장의 나머지에서는 SMP 하드웨어에 대해서 이야기하고, 병렬 프로그램 프로세스 사이에서 메모리를 공유하는 기본적인 리눅스 메커니즘을 살펴보고, 원자성(atomicity), 휘발성(volatility), 락(lock), 캐시 라인(cache line)에 대해서 아주 간단하게 알아보고, 마지막으로 여러가지 공유 메모리 병렬 처리 라이브러리들에 대해 약간의 조언을 하도록 한다.

2.1 SMP 하드웨어(Hardware)

SMP 시스템들은 여러해 전부터 사용되어 왔지만, 얼마전까지만해도 기계마다 기본적인 기능들을 서로 다르게 구현하는 경향이 있어서, 운영체제에서 SMP를 지원하는 것이 호환성이 없었다. 이런 문제를 종식시킨 것은 인텔에서 발표한 다중프로세서 규약(Multiprocessor Specification, 간단히 줄여서 **MPS**라고 한다)이다. MPS 1.4 규약은 `<http://www.intel.com/design/pro/datashts/242016.htm>`에서 PDF 파일 형식으로 된 문서로 구할 수 있으며, `<http://support.intel.com/oem_developer/ial/support/9300.HTM>`에서 MPS 1.1 규약에 대한 개요를 볼 수 있지만, 인텔이 자신의 WWW 사이트를 종종 개편을 하기 때문에 이 주소는 바뀌었을 수도 있다. 많은 *제작들* `<http://www.uruk.org/~erich/mps-hw.html>`은 4개의 프로세서까지 지원하는 MPS 호환 시스템들을 만들고 있지만, 이론적으로 MPS는 더 많은 프로세서들을 지원할 수 있다.

MPS가 아니면서 IA32(인텔 32비트 CPU)가 아닌 시스템 중에서 SMP 리눅스가 지원하는 시스템으로는 Sun4m 다중프로세서 SPARC 시스템이 유일하다. SMP 리눅스는 인텔 MPS 1.1과 1.4 호환 시스템을 대부분 지원하며, 16개까지의 486DX, Pentium, Pentium MMX, Pentium Pro, Pentium II 프로세서를 지원한다. 지원하지 않는 IA32 프로세서로는 인텔 386, 486SX/SLC 프로세서와 (부동 소숫점 연산 하드웨어가 없으면 SMP 기계에 맞지 않는다), AMD와 Cyrix의 프로세서들이다 (이들 프로세서는 다른 SMP 지원 칩들을 필요로 하는데, 이 글을 쓰고 있을 때 아직 이들 칩들은 나와있지 않았다).

MPS 호환 시스템들의 성능은 천차만별로 달라질 수 있다는 점은 꼭 이해하고 넘어가야 한다. 일반적인 예상대로 성능의 차이를 나타내는 요인 중의 하나는 프로세서 속도이다. 대체로 좀 더 빠른 클럭의 프로세서을 사용하면 더 빠른 시스템이 되며, Pentium Pro 프로세서를 사용한 시스템이 Pentium 프로세서를 이용하는 시스템보다 빠른 경향이 있다. 그렇지만 MPS에서는 공유 메모리(shared memory)를 하드웨어적으로 어떻게 구현해야 하는지는 명시하지 않고 있다. 단지 소프트웨어적인 관점에서 공유 메모리가 어떻게 동작해야 하는지만 명시하고 있을 뿐이다. 그래서 구현하고 있는 공유 메모리 방식이 SMP 리눅스의 특징과 특정 프로그램의 특징에 어떻게 맞아들어가느냐에 따라서 성능이 달라질 수 있다.

MPS 호환 시스템들의 차이는 우선 물리적으로 공유 메모리에 접근하는 것을 어떻게 구현하느냐에서 나타난다.

2.1.1 각 프로세서가 독자적인 L2 캐시를 가지는가?(Does each processor have its own L2 cache?)

일부 MPS Pentium 시스템과, 모든 MPS Pentium Pro와 Pentium II 시스템은 독자적인 L2 캐시를 가지고 있다. (L2 캐시는 Pentium Pro나 Pentium II 모듈에 들어있다) 일반적으로 독자적인 L2 캐시를 사용하면 처리 속도를 최대화할 수 있다고 알려져 있지만, 리눅스에서는 명백하게 그런 것은 아니다. 이를 혼란하게 하는 주된 이유는, 현재의 리눅스 스케줄러가 각 프로세스를 똑같은 프로세서에서 실행되게 하는 **프로세서 친화력(processor infinity)**의 개념을 따르지는 않기 때문이다. 프로세스가 실행되는 프로세서는 금방 바뀔수 있다. 이 문제는 최근에 "프로세서 결합(processor binding)"이라는 제목으로 SMP 리눅스 개발 공동체에서 토론된 적이 있다. 프로세서 친화력 없이 별도의 L2 캐시를 갖게되면, 어떤 프로세스가 이전에 실행

되던 프로세서가 아닌 다른 프로세서에서 시간을 할당받아 실행되는 경우 상당한 오버헤드를 초래할 수 있다.

상대적으로 값이 싼 상당수의 시스템들은 두개의 Pentium 프로세서가 하나의 L2 캐시를 공유하도록 만들어져 있다. 이 방식의 안좋은 점은 두개의 프로세서가 서로 캐시를 사이에 두고 경쟁을 해야한다는 것으로, 특히 여러개의 서로 독립적인 프로그램을 실행하는 경우 성능이 현저하게 떨어진다는 것이다. 좋은 점은 많은 병렬 프로그램들에게 있어서 두 개의 프로세서가 공유 메모리의 똑같은 라인에 접근하는 경우 하나만이 이를 캐시에 가져오면 되어, 버스를 둘러싼 경쟁을 피할 수 있어서 캐시를 공유하는게 실질적으로 도움이 된다는 것이다. 또한 프로세서 친화력을 적용하지 않는 경우 L2 캐시를 공유하는 것이 피해가 더 적다. 따라서 병렬 프로그램의 경우 L2 캐시를 공유하는 것이 일반적으로 생각하는 것만큼 나쁘지는 않다.

두개의 Pentium 프로세서가 256K 캐시를 공유하는 시스템을 사용해본 경험에 따르면, 필요한 커널 작업의 정도에 따라서 시스템의 성능이 상당히 크게 달라졌다. 최악의 경우 속도가 1.2배 정도밖에 빨라지지 않았지만, "데이터를 가져오는 것은 공유하는(shared fetch)" 효과를 제대로 이용하는 계산 중심적인 SPMD 스타일의 코드를 사용하였을 때 2.1 배까지 빨라지는 것을 보기도 하였다.

2.1.2 버스 설정(Bus configuration)?

먼저 이야기할 것은 요즘에 나오는 대부분의 시스템들은, 프로세서에 하나 이상의 PCI 버스가 연결되어 있고, 이는 또다시 브릿지(bridge)를 통하여 하나 이상의 ISA/EISA 버스에 연결되어 있다는 것이다. 브릿지를 통하는 경우 대기시간(latency)이 늘어나게 되고, EISA나 ISA는 일반적으로 PCI에 비해서 낮은 대역폭을 제공하기 때문에 (ISA가 제일 낮다), 디스크 드라이브나, 비디오 카드, 다른 고성능 장치들은 PCI 버스 인터페이스를 통하여 연결되어야 한다.

PCI 버스가 하나밖에 없더라도 계산 중심적인 병렬 프로그램의 경우 MPS 시스템은 괜찮은 성능개선 효과를 보여준다. 하지만 I/O 처리속도는 하나의 프로세서를 사용할 때보다 더 나아지지 않으며, 프로세서들이 버스를 사이에 두고 경쟁을 하기 때문에 아마도 성능이 조금 떨어지게 될 것이다. 따라서 I/O 속도를 높이고 싶다면 여러개의 독자적인 PCI 버스와 I/O 콘트롤러(예를 들어 여러개의 SCSI 체인들)를 가지는 MPS 시스템을 구입하는 것이 좋다. 이 때 SMP 리눅스에서 이들 시스템을 지원하는지 조심스럽게 살펴보아야 한다. 또한 현재의 SMP 리눅스에서는 어느 순간이든 하나의 프로세서만이 커널모드에 있을 수 있기 때문에, I/O 처리를 할 때 커널에서 소요하는 시간이 적은 I/O 콘트롤러를 사용해야 한다는 점을 명심해야 한다. 정말 고성능을 원한다면 시스템 콜(system call)을 통하지 않고, 사용자 프로세스가 직접 장치로 I/O를 하는 것도 고려해 볼 필요가 있다. 이는 생각만큼 어렵지도 않고, 안정성을 해치지도 않는다 (3.3 장에서 기본 기술에 대해서 설명하고 있다).

버스 속도와 프로세서 클럭 속도의 관계를 살펴보는 것도 매우 중요하다. 지난 몇해동안 이들 사이의 관계는 불명확하게 이해되어 왔다. 대부분의 시스템이 지금은 똑같은 PCI 클럭 속도를 사용하고 있지만, 더 빠른 클럭 속도의 프로세서가 더 느린 버스 클럭과 쌍을 이루는 것은 드문 일이 아니다. 이의 고전적인 예로, 일반적으로 Pentium 133은 Pentium 150보다 더 빠른 버스를 사용하였고, 다양한 벤치마크에서 특이한 결과를 나타냈다. 이런 효과는 SMP 시스템에서 더욱 증폭된다. 이는 버스 클럭 속도를 빠르게 하는 것보다도 더 중요한 문제이다.

2.1.3 메모리 중첩과 DRAM 기술(Memory interleaving and DRAM technologies)?

메모리 중첩은 실제로 MPS와는 아무런 일도 같이 하지 않는다. 그러나 MPS 시스템에서 이것이 종종 언급되는 것을 볼 수 있는데, 이는 이들 시스템이 대체로 메모리 대역폭을 더 많이 필요로 하기 때문이다. 기본적으로 2-way나 4-way 중첩은 RAM에 블럭 접근을 할 때, 이것이 하나가 아니라 여러개의 RAM 뱅크(bank)를 사용하여 이루어지도록 RAM을 조직화한다. 이는 더 높은 메모리 접근 대역폭을 제공하게 되는데, 특히 캐시 라인(cache line) 읽기나 쓰기에 있어서 더욱 그러하다.

이것의 효과에 대해서는 그다지 명쾌하지 않은데, EDO DRAM이나 여러가지 다른 메모리 기술들은 이와 비슷한 종류의 연산 속도를 향상시키기 때문이다. `<http://www.pcguide.com/ref/ram/tech.htm>`에서 DRAM 기술에 대해 무척 잘 정리되어있는 개요를 볼 수 있다.

그렇다면, 예를 들어 2-way의 중첩되는 EDO DRAM을 쓰는 것이 중첩을 사용하지 않는 SDRAM을 쓰는 것보다 더 좋은가? 이것은 매우 훌륭한 질문이며, 그 대답은 간단하지 않다. 왜냐하면 중첩기술이나 다른 흥미있는 기술들은 대체로 비싸기 때문이다. 여기에 들어가는 똑같은 돈을 보통 메모리에 투자한다면 훨씬 많은 양의 메인 메모리를 사용할 수 있을 것이다. 가장 느린 DRAM을 사용한다 하더라도 디스크를 이용한 가상 메모리보다는 훨씬 빠르다.

2.2 공유 메모리 프로그래밍에 대한 소개

SMP에서 병렬처리를 사용하는 것이 충분히 할만한 것이라고 결정을 내렸다면, 이제 어디서부터 시작하는 게 좋을까? 그럼, 공유 메모리 통신이 실제로 동작하는 방식에 대해서 조금 더 배우는 것으로 그 첫발을 내딛어보도록 하자.

얼핏 생각하면 공유 메모리 통신이란 하나의 프로세서가 메모리에 값을 저장하면, 다른 프로세서가 이를 읽어들이는 것이라고 생각할 수도 있다. 하지만 불행히도 그렇게 간단하지만은 않다. 예를 들어, 프로세스와 프로세서 사이의 관계가 무척 복잡하게 얽혀 되어있다고 하자. 프로세서의 갯수보다 현재 동작하는 프로세스의 수가 적다고 하더라도 그렇고, 그 반대의 경우도 마찬가지다. 이 장의 남은 부분에서는 특별히 신경쓰지 않으면 심각한 문제를 야기할 수 있는 중요한 논점들 - 무엇을 공유할 것인지 판단하는데 사용하는 두가지 서로 다른 모델과, 원자성(atomicity) 논점, 휘발성(volatility) 개념과 하드웨어 락(lock) 명령, 캐시 라인(cache line) 효과, 그리고 리눅스 스케줄러 논점 - 을 간단히 요약하도록 하겠다.

2.2.1 모두 공유하기 대 일부를 공유하기(Shared Everything Vs. Shared Something)

공유 메모리 프로그래밍에서는 일반적으로 **모두 공유하기**와 **일부를 공유하기**라는 두가지의 근본적으로 서로 다른 모델을 사용한다 . 이 두가지 모델은 모두 프로세서들이 공유메모리로 데이터를 쓰고, 공유메모리에서 데이터를 읽어들임으로써 통신을 할 수 있게 한다. 두 모델의 다른점은, 모두 공유하는 모델에서는 모든 자료구조를 공유메모리에 두는 반면에, 일부를 공유하는 모델에서는 사용자가 공유할 자료구조와 하나의 프로세서에 **국한되는** 자료구조를 명시적으로 지정한다는 것이다.

어떤 공유메모리 모델을 사용할 것인가? 이는 종교에 대한 질문과 비슷하다. 많은 사람들은 자료구조를 선언할 때 이것을 공유할 것인지 따로 구별할 필요가 없기 때문에, 모두 공유하는 모델을 좋아한다. 이 때는 동시에 하나의 프로세스(프로세서)만이 자료에 접근할 수 있도록, 충돌을 일으킬 수 있는 공유하는 자료에 접근하는 코드 주위에 락(lock)을 걸기만 하면 된다. 그렇지만 이것 역시 말처럼 간단하진 않다. 그래서 많은 사람들은 일부만을 공유하는 모델이 가져다주는 상대적인 안전성을 더 선호하기도 한다.

모두 공유하기(Shared Everything) 모두 공유하는 방식의 장점은 이미 만들어져 있는 순차적인 프로그램을 선택하여 쉽게 모두 공유하는 병렬 프로그램으로 변환할 수 있다는 것이다. 여기서는 어떤 자료가 다른 프로세서에서 접근할 수 있는 것인지 먼저 판단해야 할 필요가 없다.

간단하게 살펴보면, 모든걸 공유하는 방식의 가장 큰 문제점은 하나의 프로세서가 취한 행동이 다른 프로세서들에게 영향을 미칠 수 있다는 것이다. 이 문제는 두가지 방향으로 나타난다 :

- 많은 라이브러리들은 공유할 수 없는 자료구조들을 사용한다. 예를 들어, UNIX에서 대부분의 함수들은 `errno`라는 변수에다가 에러코드를 담아 돌려준다. 만약 모두 공유하는 두 개의 프로세스가 여러가지 함수를 부른다면, 이들은 똑같은 `errno` 변수를 공유하기 때문에 서로 간섭을 일으키게 될 것이다. 비록 지금은 `errno` 문제를 해결한 라이브러리가 있긴 하지만, 이와 비슷한 문제는 대부분의 라이브러리에 여전히 남아 있다. 예를 들어, 미리 특별한 주의를 기울이지 않고, 모두 공유하는 여러개의 프로세스들이 X 라이브러리의 함수들을 호출한다면, X 라이브러리는 제대로 동작하지 않을 것이다.
- 일반적으로 포인터를 잘못 사용하거나, 배열에서 인덱스를 잘못 지정한 경우, 최악의 결과로 이 코드를 수행하던 프로세스가 죽기도 한다. 이때 core 파일을 만들어 무슨 일이 일어났는지 단서를 제공해 주기도 한다. 모두 공유하는 병렬처리에서는 이런 잘못된 접근이 발생하면 *다른 프로세스까지도 죽게 할* 가능성이 커서, 지역화(localize)를 하거나 에러를 고치는 것을 거의 불가능하게 만든다.

이런 종류의 문제는 일부를 공유하는 방식에서는 흔히 일어나진 않는다. 왜냐하면 명백하게 지정한 자료구조만이 공유되기 때문이다. 그리고, 모두 공유하는 방식은 모든 프로세서가 완전히 똑같은 메모리 이미지를 실행하는 경우에만 동작한다는 것은 당연한 일이다. 즉, 여러개의 서로 다른 코드 이미지들 사이에서는 모두 공유하는 방식을 사용할 수 없다 (다르게 말하면, SPMD만을 사용할 수 있지, 일반적인 MIMD는 사용할 수 없다).

모두 공유하는 방식을 지원하는 가장 일반적인 유형은 **쓰레드 라이브러리(threads library)**이다. *쓰레드* <http://liinwww.ira.uka.de/bibliography/Os/threads.html>는 본래, 대체로 일반적인 UNIX 프로세스와는 다르게 스케줄이 이루어지고, 가장 중요한 점으로 동일한 메모리 맵에 접근할 수 있는 "가벼운" 프로세스이다. POSIX *Pthreads* <http://www.mit.edu:8001/people/proven/pthreads.html>패키지는 여러 포팅 프로젝트에서 촛점을 받아 왔었다. 여기서 중요한 질문은, 이들 포팅중의 어떤 것들이 실제로 프로그램에 있는 쓰레드들을 SMP 리눅스에서 병렬로 실행할 수 있느냐이다 (이상적으로, 각 쓰레드마다 하나의 프로세서를). POSIX API는 이를 요구하지 않으며, <http://www.aa.net/~mtp/PCthreads.html>같은 버전에서는 분명하게 병렬 쓰레스 실행을 구현하지 않고 있다 - 프로그램의 모든 쓰레드들은 하나의 리눅스 프로세스 안에 들어있다.

SMP 리눅스에서의 병렬처리를 지원한 첫번째 쓰레드 라이브러리는 지금은 한물간 bb_threads 라이브러리로, <ftp://caliban.physics.utoronto.ca/pub/linux/>에서 구할 수 있다. 이는 리눅스의 `clone()`함수를 사용하여, 독자적으로 스케줄되며, 하나의 주소공간을 공유하는, 새로운 리눅스 프로세스를 생성(fork)하는 매우 작은 라이브러리이다. SMP 리눅스 기계는 각 "쓰레드들"이 완전한 리눅스 프로세스이기 때문에 여러개의 이들 "쓰레드들"을 병렬로 실행할 수 있다. 대신 이의 댓가로 다른 운영체제의 일부 쓰레드 라이브러리들이 제공하는 것과 같은 "가벼운" 스케줄링 제어를 할 수 없다. 이 라이브러리는 새로운 메모리 조각을 각 쓰레드의 스택으로 할당하고, 락(lock)의 배열(mutex 개체들)들을 원자적으로 접근할 수 있는 함수를, C로 포장된 어셈블리 코드를 조금 사용하여 제공하고 있다. 문서는 `README`와 간단한 예제 프로그램으로 구성되어 있다.

좀더 최근에 clone()을 사용하는 POSIX 쓰레드 버전이 개발되었다. 이 라이브러리는 *LinuxThreads* <http://pauillac.inria.fr/~xleroy/linuxthreads/>로, SMP 리눅스에서 사람들이 가장 선호하는 모두 공유하는 라이브러리이다. POSIX 쓰레드들도 문서화가 잘되어있고, *LinuxThreads README* <http://pauillac.inria.fr/~xleroy/linuxthreads/README>와 *LinuxThreads FAQ* <http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html> 역시 매우 잘되어 있다. 지금의 주요한 문제는 POSIX 쓰레드를 제대로 하려면 이를 자세하게 알아야한다는 것이고, LinuxThreads는 아직은 계속해서 작업중이라는 것이다. 또한 POSIX 쓰레드 표준이 표준화 과정에서 계속 발전되고 있어, 이미 바뀐 예전 버전의 표준에 맞춰 프로그램을 작성하지 않도록 주의를 기울여야 한다는 것 역시 문제이다.

일부를 공유하기(Shared Something) 일부를 공유하는 것은 정말로 "공유할 필요가 있는 것만을 공유하는" 것이다. 이 접근법은 각 프로세서의 메모리 맵의 똑같은 위치에 공유데이터가 할당되게 하는 것에 유의한다면 일반적인 MIMD(SPMD가 아니라) 용으로 동작하게 된다. 더 중요한 특징은, 일부를 공유하는 방식은 성능을 예측하고 조율하며, 코드를 디버깅하는 것 등을 쉽게 만들어준다는 것이다. 유일한 문제로는 :

- 사전에 무엇이 정말로 공유할 필요가 있는지 아는게 힘들다.
- 공유 메모리에 객체를 실제로 할당하는 것은 골치아픈 작업이다. 특히 스택에 할당되던 객체인 경우 더욱 그렇다. 예를 들어, 공유 데이터를 별도의 메모리 영역에 할당을 해야할 필요가 있는 경우가 있는데, 이 때는 별도의 메모리 할당 함수를 사용하고, 각 프로세스가 이를 참조할 때 또다른 포인터를 사용해야 한다.

현재 리눅스 프로세스 그룹들이 독자적인 메모리 공간을 가지면서, 상대적으로 작은 메모리 영역만을 함께 공유하게 하는데에는 두개의 유사한 방식을 사용한다. 리눅스 시스템을 설정할 때 바보같이 "System V IPC"를 빼버리지 않았다면, 리눅스는 "System V 공유 메모리"라는 다른 시스템 사이에서도 호환성이 있는 방식을 제공한다. 다른 방식은 `mmap()` 시스템 콜을 통하여 메모리 매핑(memory mapping) 기능을 사용하는 것이다. 이는 구현방식이 UNIX 시스템마다 크게 차이가 난다. 이들 호출에 대해서는 매뉴얼 페이지들에서 배울 수 있다. 그리고 2.5장과 2.6장에 나오는 개괄은 이를 처음 시작할 때 도움이 될 것이다.

2.2.2 원자성과 순서(Atomicity And Ordering)

위의 두가지 모델 중 어떤 것을 사용하더라도 결과는 매우 비슷하다. 여러분은 자신이 만든 병렬 프로그램에 들어 있는 모든 프로세스들이 접근하여 읽고 쓸수 있는 메모리 조각에 대한 포인터를 얻게 된다. 이 말은 내가 만든 병렬 프로그램이 공유 메모리 객체들을 마치 보통의 지역 메모리에 있는 것처럼 접근할 수 있다는 것을 의미하지는 않는다.

원자성이란 한 객체에 대한 작업이 쪼개지지 않고, 중단될 수 없는 일련의 과정으로 이루어지는 것을 가리키는 개념이다. 불행히도, 공유 메모리에 대한 접근은 공유 메모리에 있는 자료에 대한 모든 작업이 원자적으로 이루어진다는 것을 내포하진 않는다. 미리 특별한 주의를 기울지지 않는다면, 버스(bus)에서 단 한번만에 처리가 이루어지는 간단한 읽기/쓰기 연산만이 (즉, 정렬이 된 8, 16, 32 비트 연산이지, 정렬이 안되어 있거나 64 비트 연산은 아니다) 원자성을 가진다. 더욱 나쁜 것은, GCC같이 "똑똑한" 컴파일러는 최적화를 통해 메모리 작업을 제거하여, 한 프로세서가 한 일을 다른 프로세서에서 볼 수 없게 만들어버리기도 한다. 다행히도, 이들 문제들은 모두 고칠 수 있다... 접근 효율성(access efficiency)과 캐시라인 크기(cache line size) 사이의 관계만 걱정거리로 남겨두고서 말이다.

그렇지만 이들 논점에 대해서 토론하기 전에, 이들은 모두 각 프로세서에서의 메모리 참조가 코딩한 순서대로 이루어지고 있다고 가정하고 있다는 것을 지적할 필요가 있다. Pentium은 그렇게 하고 있지만, 앞으로 나올 인텔의 프로세서들은 그렇지 않을 수도 있다는 것도 기억하기 바란다. 따라서, 앞으로 나올 프로세서에 대비하여, 공유 메모리에 접근하는 코드 주위를, 모든 미결된 메모리 접근을 완료하여 메모리 접근이 차례대로 이루어지도록 하는 명령어로 둘러싸야 할 필요가 있다는 것을 깊이 새기길 바란다. `CPUID` 명령어는 이런 부수효과(side-effect)를 위해 예약되어 있는 것이다.

2.2.3 휘발성(Volatility)

GCC 옵티마이저(optimizer)가 공유 메모리 객체의 값을 레지스터에 버퍼링하는 것을 막으려면, 공유 메모리에 있는 모든 객체들을 `volatile` 속성을 가지도록 선언해야 한다. 이렇게 하면, 한번의 접근만으로 이루어지는 모든 공유 객체의 읽기/쓰기는 원자적으로 일어나게 된다. 예를 들어, p가 정수에 대한 포인터이고, 이것이 가리키고 있는 정수가 공유 메모리에 있다고 하자. ANSI C에서는 이를 다음과 같이 정의할 수 있다.

```
volatile int * volatile p;
```

이 코드에서, 첫번째 `volatile`는 `p`가 가리키는 `int` 값을 말하며, 두번째 `volatile`는 포인터 그 자체를 말한다. 물론 이는 귀찮은 작업이지만, GCC가 매우 강력한 최적화를 수행할 수 있도록 하기 위해서 치러야 하는 것이다. 적어도 이론적으로는, GCC에 `-traditional` 옵션을 주는 것으로도, 몇가지 최적화를 희생하는 대신 올바른 코드를 만들어내는데에는 충분하다. 왜냐하면 ANSI K&R C 이전에는 모든 변수는 따로 `register`라고 지정하지 않은 이상 모두 volatile이었기 때문이다. 여전히 GCC로 `cc -O6`와 같이 컴파일을 하고, 필요한 것에만 volatile이라고 지정할 수도 있다.

모든 프로세서의 레지스터를 수정하는 것으로 표시되어 있는 어셈블리어 락(lock)을 사용하면, GCC가 모든 변수들을 다 내보내서(flush), `volatile`이라고 선언함으로써 발생하는 "비효율적인" 코드들을 피할 수 있게 하는 효과가 있다는 소문이 있어왔다. 이런 방법은 GCC 2.7.0을 사용하는 경우, 정적으로 할당되는 전역변수에 대해서는 제대로 동작하는 것처럼 보인다... 그렇지만, 이런 행동은 ANSI C 표준에서는 필요하지 않다. 더 나쁜 것은 읽기 접근만을 하는 다른 프로세스들은 변수 값을 영원히 레지스터에 버퍼링을 할 수 있어서, 공유 메모리의 값이 실제로 변하는 것을 절대로 알아차리지 못할 수도 있다. 요약하면, 하고 싶은대로 해도 좋지만, `volatile`라고 지정한 변수만이 제대로 동작한다는 것을 *보장할 수 있다*.

일반 변수에도 `volatile` 속성을 암시하는 형변환(type cast)을 사용하여 volatile 접근을 할 수 있다. 예를 들어, 보통의 `int i;`는 `*((volatile int *) &i);`같이 선언하여 volatile로 접근할 수 있다. 이렇게 하면 휘발성(volatility)이 필요한 경우에만, 이런 오버헤드를 사용하도록 할 수 있다.

2.2.4 락(Locks)

`++i;`는 항상 공유 메모리에 있는 변수 `i`에 1을 더한다고 생각해왔다면, 다음 이야기는 조금은 놀랍고 당황스러울지도 모르겠다. 하나의 명령으로 코딩을 했다고 하더라도, 값을 읽고 결과를 쓰는 것은 별도의 메모리 처리(transaction)을 통해서 이루어지며, 이 두 처리 사이에 다른 프로세서가 `i`에 접근할 수도 있다. 예를 들어, 두개의 프로세서가 모두 `++i;` 명령을 수행하였는데, 2가 증가하는게 아니라 1이 증가할 수도 있다는 것이다. 인텔 Pentium의 "구조(Architecture)와 프로그래밍 매뉴얼"에 따르면, `LOCK` 접두어는 다음에 나오는 명령어가 그것이 접근하는 메모리 위치에 대해 원자적으로 이루어지는 것을 보장하기 위해 사용된다.

```
BTS, BTR, BTC                     mem, reg/imm
XCHG                              reg, mem
XCHG                              mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR  mem, reg/imm
NOT, NEG, INC, DEC                mem
CMPXCHG, XADD
```

그렇지만, 이들 연산을 모두 사용하는 것은 그다지 좋은 생각은 아닌 것 같다. 예를 들어, `XADD`는 386에서는 존재하지도 않고, 따라서 이를 사용하는 것은 호환성의 문제를 발생시킬 수 있다.

`XCHG` 명령어는 `LOCK` 접두어가 없더라도 항상 락을 사용한다. 따라서 이 명령어는 세마포어(semaphore)나 공유 큐(shared queue)같은 고수준의 원자적인 구성체를 만드는 경우에 좋은 원자적인 연산이다. 당연히 C 코드로 GCC가 이 명령어를 만들어내도록 할 수는 없다. 대신 인라인(in-line) 어셈블리 코드를 조금 사용해야 한다. 워드(word) 크기의 volatile 객체인 *obj*와 워드 크기의 레지스터 값인 *reg*가 있다면, GCC 인라인 어셈블리 코드는 다음과 같다 :

```
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (obj)
                      :"r" (reg), "m" (obj));
```

락(lock)을 하는데 비트(bit) 연산을 사용하는 GCC 인라인 어셈블리 코드의 예제가 *bb_threads library* `<ftp://caliban.physics.utoronto.ca/pub/linux/>`라이브러리의 소스 코드에 있다.

메모리 처리(transaction)를 원자적으로 만드는 것에도 이에 따르는 비용이 있다는 것을 기억할 필요가 있다. 보통의 참조는 지역 캐시를 사용할 수 있다는 사실에 비추어보면, 락(lock)을 하는 연산은 약간의 오버헤드를 수반하고, 다른 프로세서의 메모리 활동을 지연시킬 수 있다. 락(lock) 연산을 사용하는 경우 가장 좋은 성능을 내고 싶다면 가능한 이를 적게 사용하는 것이 좋다. 더 나아가 이들 IA32 원자적인 명령어들은 다른 시스템과의 호환성이 없다.

어떤 순간이든 많아도 하나의 프로세서만이 주어진 공유 객체를 갱신하는 것을 보장하는 여러가지 동기화(synchronization) - **상호 배제(mutual exclusion)를 포함하여** - 를 구현하는데 보통의 명령어를 사용할 수 있도록 하는 여러가지 다른 접근방법이 있다. 대부분의 OS 교재에서는 이들 기법을 적어도 하나 이상씩은 다루고 있다. Abraham Silberschatz와 Peter B Galvin이 지은 *운영체제 개념(Operating System Concepts)* 4판(ISBN 0-201-50480-4)에서 이를 아주 잘 다루고 있다.

2.2.5 캐시라인 크기(Cache Line Size)

원자성에 관련된 기본적인 것으로서 SMP 성능에 큰 영향을 미칠 수 있는 것으로 캐시라인 크기가 있다. MPS 표준에는 어떤 캐시가 사용되든지 간에 참조는 일관적이어야 한다고 하고 있지만, 사실은 하나의 프로세서가 메모리의 특정 라인에 기록을 할 때, 이전 라인의 캐시된 복사본이 모두 무효화(invalidate)되거나 갱신(update)되어야 한다. 이 말은 두 개나 그 이상의 프로세서가 동시에 같은 라인의 다른 부분에 데이터를 기록하려고 하면, 상당량의 캐시와 버스 통행(traffic)이 발생할 수 있으며, 실질적으로 캐시에서 캐시로

라인을 전달하게 된다. 이 문제는 **잘못된 공유(false sharing)**라고 한다. 그 해결책은 *병렬로 접근되는 데이터가 되도록이면 각 프로세서마다 다른 캐시 라인에서 올 수 있도록 데이터를 조직화하도록 하는 것이다.*

잘못된 공유는 L2 캐시를 사용하는 시스템에서는 문제가 안될거라고 생각할 수도 있겠지만, 여전히 별도의 L1 캐시가 있다는 것을 기억하자. 캐시의 조직과 구별된 별도의 레벨의 갯수는 모두 변할 수 있지만, Pentium L1 캐시라인 크기는 32 바이트이고, 전형적인 외장형 캐시라인 크기는 256 바이트 가량이다. 두 항목의 주소가 (물리적 주소이든, 가상 주소이든) a와 b이고, 가장 큰 프로세서당 캐시라인 크기가 c이고, 이들은 모두 2의 몇 제곱승이라고 하자. 매우 엄밀하게 하면, `((int) a) & ~(c - 1)`와 `((int) b) & ~(c - 1)`이 같을 때, 두개의 참조가 똑같은 캐시라인에 존재하게 된다. 더 규칙을 간단화하면 병렬로 참조되는 공유 객체가 적어도 c 바이트가 떨어져 있다면, 이들은 다른 캐시 라인으로 매핑이 된다는 것이다.

2.2.6 리눅스 스케줄러 논점(Linux Scheduler Issues)

병렬처리에서 공유 메모리를 사용하는 전적인 이유는 OS의 오버헤드를 피하자는 것이지만, OS 오버헤드는 통신 그 자체 외의 것에서 발생하기도 한다. 우리는 이미 만들어야 할 프로세스의 갯수가 기계에 있는 프로세서의 갯수보다 같거나 작아야한다고 말했었다. 그러나 정확히 얼마나 많은 프로세스를 만들어야 할지 어떻게 결정할 수 있을까?

최고의 성능을 내려면, *여러분이 작성한 병렬 프로그램에 있는 프로세스의 갯수는, 다른 프로세서에서 계속해서 실행될 수 있는 프로세스의 갯수하고 같아야 한다.* 예를 들어, 네개의 프로세서가 있는 SMP 시스템에서 하나의 프로세스가 다른 목적으로 (예를 들어 WWW 서버) 동작하고 있다면, 여러분이 만든 병렬 프로그램은 세개의 프로세스만을 사용해야 한다. 시스템에 몇 개의 다른 프로세스들이 있는지는 `uptime` 명령에서 돌려주는 "평균 부하(load average)"를 참조하여 대강은 알 수 있다.

다른 방법으로 `renice` 명령이나 `nice()` 시스템 콜 같은 것을 이용하여 병렬 프로그램에 있는 프로세스의 우선순위(priority)를 높일 수도 있다. 우선순위를 높이려면 권한이 있어야 한다. 이 생각은 다른 프로세스를 프로세서에서 ゑご아내서 자신이 만든 프로그램이 모든 프로세서에서 계속해서 실행될 수 있게 하는 것이다. 이 방법은 `<http://luz.cs.nmt.edu/~rtlinux/>` 에 있는 실시간(real-time) 스케줄러를 제공하는 SMP 리눅스의 프로토타입(prototype) 버전을 사용하면 좀더 확실하게 달성할 수 있다

여러분이 SMP 시스템을 병렬 기계로 사용하는 유일한 사용자가 아니라면, 계속 실행하려고 하는 두 개 이상의 병렬 프로그램 사이에 충돌이 빚어질 수도 있다. 이의 표준 해결방법은 **조단위(gang) 스케줄링** - 즉 동시에 하나의 병렬 프로그램에 속하는 프로세스들만이 실행될 수 있도록 스케줄링 우선순위를 다루는 것이다. 그렇지만 하나 이상을 병렬처리하면 결과가 늦게 돌아오고, 스케줄러의 활동이 오버헤드를 더하게 된다는 것을 상기하기 바란다. 따라서, 예를 들어 네 개의 프로세서를 가진 시스템에 두 개의 프로그램을 실행한다면, 두 개의 프로세스를 각각 실행하는 것이, 두 개의 프로그램이 네개의 프로세스를 각각 사용하면서 조단위 스케줄링을 하는 것보다 더 낫다.

이런 문제를 더 꼬이게 하는 것이 하나 더 있다. 여러분이 낮에는 종일 과중하게 사용되고 있지만, 밤에는 완전히 병렬처리용으로만 사용가능한 기계에서 프로그램을 개발하고 있다고 하자. 여러분은 낮에 테스트를 하는것이 느리다는 것을 알더라도 프로그램을 작성하고 작성한 코드를 테스트하고 수정하기 위해 모든 갯수의 프로세스를 만들어 사용할 것이다. 그런데, 프로세스들이 현재 실행되고 있지 않는 (다른 프로세서에서) 다른 프로세스와 공유 메모리로 값을 전달하기만을 아무것도 하지 않고 기다리고 있다면, 이들 작업은 매우 느려질 것이다. 이와 똑같은 문제는 코드를 하나의 프로세서밖에 없는 시스템에서 개발하고 테스트하는 경우에도 발생한다.

해결책은 코드에서 다른 프로세서에서 일어나는 동작을 하염없이 기다려야 하는 부분에, 리눅스가 다른 프로세스를 실행할 수 있는 기회를 주도록 함수 호출을 집어넣는 것이다. 나는 C 매크로를 사용하는데 이를 하는 매크로를 `IDLE_ME`라고 부르고 있다. 프로그램을 테스트하기 위해서 컴파일을 할때는 `cc -DIDLE_ME=usleep(1)` 같이 하고, "제품"으로 실행할 때에는 `cc -DIDLE_ME={}`같이 컴파일을 한다. `usleep(1)`은 1/1000 초동안 프로세스가 잠들게 하여, 리눅스 스케줄러가 그 프로세서에서 다른 프로세스를 실행하도록 선택할 수 있게 한다. 프로세스의 갯수가 사용가능한 프로세서의 갯수보다 두배이상 많다면, `usleep(1)`를 사용하는 코드가 이를 사용하지 않는 코드보다 열배이상 빠르게 실행되는 것은 그리 이상

한 일이 아니다.

2.3 bb_threads

bb_threads("Bare Bones(뼈만남은)" threads) 라이브러리(`<ftp://caliban.physics.utoronto.ca/pub/linux/>`)는 리눅스 `clone()` 호출의 사용법을 보여주는 아주 간단한 라이브러리이다. tar 파일을 gzip으로 압축하면 겨우 7K 바이트밖에 되지 않는다! 이 라이브러리는 2.4장에서 설명하는 LinuxThreads 라이브러리 때문에 이제 한물간 것이 되었지만, 여전히 쓸만하고, 작고 간단하여 리눅스에서 지원하는 쓰레드의 사용법을 소개하는데에도 알맞다. 분명히 LinuxThreads용 소스코드를 보는 것보다 이 코드를 보는 것이 훨씬 덜 기죽을 것이다. 요약하면 bb_threads 라이브러리는 시작하기 좋은 지점이지만, 큰 프로젝트를 만들때에는 적당하진 않다.

bb_threads 라이브러리를 사용하는 프로그램의 기본적인 구조는 다음과 같다 :

1. 하나의 프로세스로 프로그램을 시작한다.

2. 이제 각각의 쓰레드가 필요로 하는 최대 스택의 크기를 계산해야 한다. 이를 크게 잡더라도 그다지 해가 되지 않는다 (이것은 가상 메모리가 존재하는 이유중의 하나이다). 그러나 모든 스택은 하나의 가상 주소 공간에서 나오기 때문에, 너무 크게 잡는 것도 좋은생각이 아니다. 예제에서는 64K를 사용하고 있다. 이 크기를 b 바이트로 설정하려면 `bb_threads_stacksize(b)`를 호출한다.

3. 다음 단계는 필요한 락(lock)들을 모두 초기화하는 것이다. 이 라이브러리에서 구현된 락 메커니즘은 락에 0부터 `MAX_MUTEXES`까지 숫자를 붙이는 것이다. 락 i를 초기화하려면 `bb_threads_mutexcreate(i)`를 호출한다.

4. 라이브러리 루틴을 호출하여 새로운 쓰레드를 만든다. 여기에 인자로 새로운 쓰레드가 실행할 함수와, 여기에 전달할 인자들을 넘겨준다. 인자로 *arg* 하나만 받고, 아무것도 돌려주지 않는 함수 f를 실행하는 쓰레드를 새로 만든다면, 함수 f>를 `void f(void *arg, size_t dummy)` 처럼 선언을 하고 `bb_threads_newthread(f, &arg)`함수를 부르면 된다. 하나 이상의 인자를 전달해야하는 경우 인자값들을 가지고 있는 구조체의 포인터를 넘겨주면 된다.

5. 병렬 코드를 실행한다. 락을 사용하는 `bb_threads_lock(n)`와 `bb_threads_unlock(n)` 함수를 사용할 때 주의를 기울인다 (여기서 n은 사용할 락을 지정한다). 이 라이브러리에 있는 락을 걸고 락을 해제하는 연산은 원자적인 버스-락(bus-lock) 명령어를 사용하는 매우 기본적인 스핀락(spin lock)이다. 그래서 과도한 메모리 접근 충돌을 일으킬 수 있으며, 어떤 접근도 공정하다는 것을 보증할 수 없다.

   bb_threads에 함께 딸려오는 예제 프로그램에서 보면 함수 `fnn`과 `main`에서 동시에 `printf()`를 실행하는 것을 막는데에 락을 옳바르게 사용하지 않고 있다. 이것 때문에 예제는 항상 동작하지는 않는다. 내가 이 말을하는 것은 예제 프로그램에 트집을 잡기 위해서가 아니라, 이것이 *매우 다루기 어렵다*는 것과 LinuxThreads를 사용하는 것이 조금 쉽다는 걸 강조하기 위해서이다.

6. 쓰레드가 `return` 명령을 실행하면, 이는 실제적으로 프로세스를 죽이게 된다. 그러나 지역 스택 메모리는 자동으로 할당이 해제되지 않는다. 엄밀하게 말하면 리눅스는 할당 해제를 지원하지 않으며, 메모리 공간은 자동으로 `malloc()`의 사용하지 않는 메모리 목록(free list)으로 되돌아가 추가되지 않는다. 따라서 부모 프로세스는 죽은 자식 프로세스마다 `bb_threads_cleanup(wait(NULL))`를 불러서 이 공간을 반납해야 한다.

다음 C 프로그램은 1.3장에서 설명한 알고리즘을 사용하여, 두개의 bb_threads 쓰레드를 이용해서 파이(pi)의 근사치를 계산한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include "bb_threads.h"

volatile double pi = 0.0;
volatile int intervals;
volatile int pids[2];      /* Unix PIDs of threads */

void
do_pi(void *data, size_t len)
{
  register double width, localsum;
  register int i;
  register int iproc = (getpid() != pids[0]);

  /* set width */
  width = 1.0 / intervals;

  /* do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* get permission, update pi, and unlock */
  bb_threads_lock(0);
  pi += localsum;
  bb_threads_unlock(0);
}

int
main(int argc, char **argv)
{
  /* get the number of intervals */
  intervals = atoi(argv[1]);

  /* set stack size and create lock... */
  bb_threads_stacksize(65536);
  bb_threads_mutexcreate(0);

  /* make two threads... */
  pids[0] = bb_threads_newthread(do_pi, NULL);
  pids[1] = bb_threads_newthread(do_pi, NULL);

  /* cleanup after two threads (really a barrier sync) */
  bb_threads_cleanup(wait(NULL));
  bb_threads_cleanup(wait(NULL));
```

```
  /* print the result */
  printf("Estimation of pi is %f\n", pi);

  /* check-out */
  exit(0);
}
```

2.4 LinuxThreads

LinuxThreads <http://pauillac.inria.fr/~xleroy/linuxthreads/> 는 POSIX 1003.1c 쓰레드 표준에 따라 "모두 공유하는" 방식을 완전하고 튼튼하게 구현한 것이다. 다른 POSIX 쓰레드를 포팅한 것과는 달리, LinuxThreads는 bb_threads에서 사용한 것과 똑같은 리눅스 커널의 쓰레드(`clone()`)를 사용한다. POSIX와 호환된다는 것은 다른 시스템에서 만든 상당수의 쓰레드 프로그램들을 상대적으로 쉽게 포팅할 수 있으며, 참고할 수 있는 다양한 예제가 있다는 것을 의미한다. 간단히 말해, LinuxThreads는 리눅스에서 방대한 규모의 쓰레드 프로그램을 개발할 때 사용할 수 있는 확실한 쓰레드 패키지이다.

LinuxThreads 라이브러리를 사용하는 기본 프로그램 구조는 다음과 같다 :

1. 하나의 프로세스로 프로그램을 시작한다.
2. 다음 단계는 필요한 락(lock)들을 모두 초기화하는 것이다. 숫자로 구별되는 bb_threads 락과는 달리 POSIX 락들은 `pthread_mutex_t`타입(type)의 변수로 선언된다. `pthread_mutex_init(&lock,val)`함수를 이용하여 필요한 각각의 락들을 초기화한다.
3. bb_threads에서처럼 새로운 쓰레드를 만들려면 라이브러리 루틴을 호출해야 한다. 여기서 인자는 새로운 쓰레드가 실행할 함수와, 여기에 전달할 인자들이다. 그렇지만 POSIX에서는 사용자가 각 쓰레드를 구별할 수 있도록 `pthread_t` 타입의 변수를 정의하는 것이 필요하다. `f()` 함수를 실행하는 `pthread_t` 쓰레드를 생성하려면 `pthread_create(&thread,NULL,f,&arg)`를 호출한다.
4. 병렬 코드를 실행한다. 필요한 경우에 `pthread_mutex_lock(&lock)`와 `pthread_mutex_unlock(&lock)`를 호출하도록 주의한다.
5. `pthread_join(thread,&retval)` 함수를 이용하여 각 쓰레드가 끝난후에 마무리를 한다.
6. C 코드를 컴파일할 때 `-D_REENTRANT` 옵션을 추가한다.

다음은 LinuxThreads를사용하여 파이(pi)를 계산하는 병렬프로그램의 예이다. 1.3장에서 사용한 알고리즘을 사용하였고, bb_threads 예제에서처럼 두개의 쓰레드가 병렬로 실행된다.

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

volatile double pi = 0.0;  /* Approximation to pi (shared) */
pthread_mutex_t pi_lock;   /* Lock for above */
volatile double intervals; /* How many intervals? */

void *
process(void *arg)
{
  register double width, localsum;
  register int i;
```

```
  register int iproc = (*((char *) arg) - '0');

  /* Set width */
  width = 1.0 / intervals;

  /* Do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* Lock pi for update, update it, and unlock */
  pthread_mutex_lock(&pi_lock);
  pi += localsum;
  pthread_mutex_unlock(&pi_lock);

  return(NULL);
}

int
main(int argc, char **argv)
{
  pthread_t thread0, thread1;
  void * retval;

  /* Get the number of intervals */
  intervals = atoi(argv[1]);

  /* Initialize the lock on pi */
  pthread_mutex_init(&pi_lock, NULL);

  /* Make the two threads */
  if (pthread_create(&thread0, NULL, process, "0") ||
      pthread_create(&thread1, NULL, process, "1")) {
    fprintf(stderr, "%s: cannot make thread\n", argv[0]);
    exit(1);
  }

  /* Join (collapse) the two threads */
  if (pthread_join(thread0, &retval) ||
      pthread_join(thread1, &retval)) {
    fprintf(stderr, "%s: thread join failed\n", argv[0]);
    exit(1);
  }

  /* Print the result */
  printf("Estimation of pi is %f\n", pi);
```

```
  /* Check-out */
  exit(0);
}
```

2.5 System V 공유 메모리

시스템 V IPC(Inter-Process Communication, 프로세스간 통신)는 메시지 큐(message queue)와 세마포어(semaphore), 공유 메모리(shared memory) 메커니즘을 제공하는 여러가지 시스템 콜을 지원한다. 물론 이들 메커니즘은 원래 하나의 프로세서를 사용하는 시스템에서 여러개의 프로세스들이 통신을 하는데 사용하기 위해서 만들어졌다. 그렇지만, 이 말은 SMP 리눅스에서 프로세스가 어떤 프로세서에서 실행되고 있든지 간에 프로세스간 통신에서도 제대로 동작해야 한다는 의미를 내포하고 있다.

이들 시스템 콜이 어떻게 사용되는지 살펴보기 전에, 시스템 V IPC 호출이 세마포어나 메시지 전달같은 일을 위해 존재하긴 하지만, 이를 사용해선 안된다는 것을 이해하는 것이 중요하다. 왜 안되느냐? 이들 함수들은 일반적으로 느리고 SMP 리눅스에서는 직렬화(serialize)되어 있다. 이정도면 충분하겠다.

공유 메모리 영역으로의 접근을 공유하는 프로세스 그룹을 만드는 기본적인 과정은 다음과 같다 :

1. 하나의 프로세스로 프로그램을 시작한다.

2. 대체로 여러분은 실행되는 각각의 병렬 프로그램이 자신만의 공유 메모리 영역을 가지기를 바랄 것이다. 따라서 `shmget()` 함수를 불러 원하는 크기만큼의 새로운 영역을 만들어야 한다. 이 호출은 이미 존재하는 공유 메모리 영역의 ID를 얻는데에도 사용할 수 있다. 어떤 경우이든, 돌아오는 값은 공유 메모리 영역의 ID이거나 에러가 발생한 경우 -1이다. 예를 들어, b 바이트 크기의 공유 메모리 영역을 만든다면, `shmid = shmget(IPC_PRIVATE, b, (IPC_CREAT | 0666))` 같이 사용할 수 있다.

3. 다음 단계는 이 공유 메모리 영역을 이 프로세스에 연결하는(attach) 것이다. 말 그대로 이 메모리를 이 프로세스의 가상 메모리 맵에 추가하는 것이다. 프로그래머는 `shamt()` 함수 호출에서 메모리 영역이 나타날 가상 주소를 지정할 수 있지만, 선택한 주소는 페이지 경계(boundary)에 따라 정렬(align)이 되어 있어야 하며 (즉, `getpagesize()`에서 돌려주는 페이지 크기 - 보통은 4096 바이트이다 - 의 배수여야 한다), 이는 이 주소에 이미 존재하던 어떤 메모리이든지간에 매핑을 덮어써버린다. 따라서, 이보다는 시스템이 주소를 고를 수 있도록 하는 것이 더 선호된다. 어떤 경우이든, 돌아오는 값은 매핑이 된 세그먼트가 시작하는 가상주소에 대한 포인터이다. 코드는 `shmptr = shmat(shmid, 0, 0)`과 같은 형태이다.

   모든 공유 변수들을 구조체의 멤버로 선언하고 *shmptr*을 이 구조체에 대한 포인터로 선언함으로써 간단하게 모든 정적 변수를 공유 메모리 영역으로 할당할 수 있다. 이 기법을 이용하여, 공유 변수 x는 *shmptr*->x로 접근할 수 있다.

4. 공유 메모리 영역을 사용하는 마지막 프로세스가 종료하거나 이 영역에서 떨어져나오면(detach) 이 공유 메모리 영역을 없애야한다. 이 기본 행동을 설정하려면 `shmctl()` 함수를 부를 필요가 있다. 코드는 `shmctl(shmid, IPC_RMID, 0)`과 같은 형태로 작성한다.

5. 원하는 갯수로 프로세스들을 만들려면 표준 리눅스의 `fork()` 함수를 사용한다. 각각의 프로세스는 공유 메모리 영역을 상속받게된다.

6. 프로세스가 공유 메모리 영역을 사용하는 작업을 끝마치면, 이 공유 메모리 영역으로부터 분리(detach)해야 한다. 이는 `shmdt(shmptr)`을 불러서 한다.

위에 설명한 과정에서는 몇개 안되는 시스템 호출만을 사용하지만, 일단 공유 메모리 영역이 만들어지면, 하나의 프로세스가 메모리상의 값을 바꾼 경우 자동으로 모든 프로세스에 보이게 된다. 가장 중요한 점은 각 통신 작업이 시스템 콜을 하는 오버헤드없이 이루어진다는 것이다.

다음은 시스템 V 공유 메모리 영역을 사용하는 C 프로그램의 예이다. 이 프로그램은 파이(pi)를 계산하는 것으로 1.3장에서 나온 것과 똑같은 알고리즘을 사용한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>

volatile struct shared { double pi; int lock; } *shared;

inline extern int xchg(register int reg,
volatile int * volatile obj)
{
  /* Atomic exchange instruction */
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (*obj)
                      :"r" (reg), "m" (*obj));
  return(reg);
}

main(int argc, char **argv)
{
  register double width, localsum;
  register int intervals, i;
  register int shmid;
  register int iproc = 0;;

  /* Allocate System V shared memory */
  shmid = shmget(IPC_PRIVATE,
                 sizeof(struct shared),
                 (IPC_CREAT | 0600));
  shared = ((volatile struct shared *) shmat(shmid, 0, 0));
  shmctl(shmid, IPC_RMID, 0);

  /* Initialize... */
  shared->pi = 0.0;
  shared->lock = 0;

  /* Fork a child */
  if (!fork()) ++iproc;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the local computations */
  localsum = 0;
  for (i=iproc; i<intervals; i+=2) {
    register double x = (i + 0.5) * width;
```

```
    localsum += 4.0 / (1.0 + x * x);
  }
  localsum *= width;

  /* Atomic spin lock, add, unlock... */
  while (xchg((iproc + 1), &(shared->lock))) ;
  shared->pi += localsum;
  shared->lock = 0;

  /* Terminate child (barrier sync) */
  if (iproc == 0) {
    wait(NULL);
    printf("Estimation of pi is %f\n", shared->pi);
  }

  /* Check out */
  return(0);
}
```

나는 이 예제에서 락(lock)을 구현하기 위해 IA32의 원자적인(atomic) 교환(exchange) 명령어를 사용하였다. 더 나은 성능과 호환성을 바란다면, 원자적인 버스-락(bus-lock) 명령어를 사용하지 않는 동기화 기법으로 대체하기 바란다.

현재 사용하고 있는 시스템 V IPC 기능들의 상태를 보여주는 ipcs을 기억하고 있다면, 여러분이 만든 코드를 디버깅할 때 도움이 될 것이다.

2.6 메모리 맵 호출

파일 I/O 시스템 콜을 사용하는 비용은 매우 클 수 있다. 사실, 이것이 사용자 버퍼를 사용하는 파일 I/O 라이브러리가 있는 이유이다 (getchar(), fwrite() 등). 그러나 사용자 버퍼는 여러개의 프로세스가 똑같은 쓰기 가능한 파일에 접근하고 있다면 사용할 수 없으며, 사용자 버퍼를 관리하는 오버헤드도 꽤 크다. BSD UNIX에서는 파일의 일부를 사용자 메모리로 매핑하여 본질적으로 가상 메모리 페이징 메커니즘을 통해 갱신을 하도록 할 수 있는 시스템 콜을 추가하여 이를 해결한다. 이와 똑같은 메커니즘이 몇년전 Sequent에서 만든 시스템에서 공유 메모리 병렬처리 지원의 기반으로 사용되었다. (아주 오래된) man 페이지에 몇가지 매우 부정적인 의견이 있음에도 불구하고, 리눅스는 기본적인 함수들 중 적어도 몇가지는 제대로 수행하는듯이 보이며, 이 시스템 콜을 여러개의 프로세스가 공유할 수 있는 무명의(anonymous) 메모리 영역으로의 매핑에 사용하는 것을 지원한다.

본질적으로 리눅스에서의 mmap() 구현은 2.5장에서 설명한 2, 3, 4번째 단계를 하나로 대체한 것이다. 무명의 공유 메모리 영역을 만들려면 :

```
shmptr =
    mmap(0,                        /* system assigns address */
         b,                        /* size of shared memory segment */
         (PROT_READ | PROT_WRITE), /* access rights, can be rwx */
         (MAP_ANON | MAP_SHARED),  /* anonymous, shared */
         0,                        /* file descriptor (not used) */
         0);                       /* file offset (not used) */
```

시스템 V 공유메모리의 shmdt() 함수와 똑같은 일을 하는 함수는 munmap()이다 :

```
munmap(shmptr, b);
```

내 생각에는 시스템 V 공유 메모리 지원 대신 mmap()을 사용하는 것이 실제로 더 낫지는 않다.

제 3 절 리눅스 시스템의 클러스터(Clusters Of Linux Systems)

이 섹션은 리눅스를 사용한 클러스터 병렬 처리의 개관을 제공하려고 시도할 것이다. 클러스터는 현재 가장 인기있는 것이자 가장 다양하다. 이것은 전통적인 워크스테이션들을 여러대 묶은 네트웍(**NOW**; network of workstations)에서 이제 막 프로세서 노드로써 리눅스 피씨들로 사용하기 시작한 커스텀 병렬 기계까지 있다. 또한 리눅스 기계들의 클러스터를 사용하는 병렬 처리를 위한 많은 소프트웨어 지원들이 있다.

3.1 왜 클러스터인가(Why A Cluster)?

클러스터 병렬 처리는 다음과 같은 중요한 장점들을 제공한다:

- 클러스터에 있는 각 기계들은 넓은 범위의 다른 컴퓨팅 어플을 사용할 수 있는 완전한 시스템이 될 수 있다. 그래서 많은 사람들이 클러스터 병렬 컴퓨팅은 사람들의 책상 위에서 할 일없이 놀고 있는 워크스테이션들의 "버려지는 시간들(wasted cycles)" 모두를 가져다 쓴다고 제안하게 되었다. 이런 시간들을 구하는 것은 실제 그렇게 쉬운 것이 아니다. 그리고 이것은 동료의 스크린 세이버를 느리게 할 것이다. 그러나 이것을 그렇게 될 수 있다.
- 네트웍 시스템들이 최근 급증하는 것은 클러스터를 만들기 위한 하드웨어 대부분이 대용량으로, 이에 대응해서 결과적으로 낮은 "상품" 가격으로, 팔리고 있다는 것을 의미한다. 클러스터 하나에 한개의 비디오 카드, 모니터, 키보드가 필요하는 사실이 좀 더 비용을 아낄 수 있도록 한다(비록 초기 리눅스 설치를 수행할 때 클러스터의 각 기계들에 이들을 옮겨 가며 써야 하지만, 일단 실행되면 전형적인 리눅스 피씨는 "콘솔"을 필요로 하지 않는다). 이와 비교해서 SMP와 부속 프로세서는 훨씬 더 작은 시장에서 단위 수행 성능에 대한 더 높은 가격을 형성해가고 있다.
- 클러스터 컴퓨팅은 *아주 커다란 시스템*까지 커질 수 있다. 현재까지 4개 이상의 프로세서들을 가지는 리눅스-호환 SMP를 찾는 것이 어려운 반면에 일반적으로 사용 가능한 네트웍 하드웨어는 16개 기계들까지 클러스터로 쉽게 묶을 수 있다. 조금만 작업하면 수백개 또는 심지어 수천개 기계들을 네트웍으로 묶을 수 있다. 사실 전체 인터넷이 하나의 커다란 클러스터로 볼 수 있다.
- 클러스터 안에서 "고장난 기계"를 교체하는 것이 패러티가 잘못된 SMP를 고치는 것에 비해서 더 단순하다는 사실은 조심스럽게 디자인된 클러스터 설정에 좀 더 높은 가용성을 제공한다. 이것은(높은 가용성은) 중요한 서비스 중단을 견딜 수 없는 특별한 어플리케이션들에 대해서 중요할 뿐만이 아니고, 단일-기계 고장이 자주 일어나는 충분한 프로세스들을 가진 시스템들의 일반적인 사용에도 중요하다. (예를 들어서 PC 고장의 평균 시간이 2년일지라도 32개 기계를 갖는 클러스터에서 적어도 한 기계가 6개월 이내에 고장날 확률은 꽤 높다.)

좋다. 클러스터는 프리이거나 싸고 아주 커질 수 있으며 가용성이 높다... 그렇다면 왜 모든 사람들이 클러스터를 사용하지 않는가? 글쎄 거기에는 다음과 같은 문제들이 존재한다:

- 아주 작은 예외들을 제외하고 네트웍 하드웨어는 병렬 처리를 위해서 고안된 것이 아니다. 전형적으로 SMP와 부속 프로세서에 비해서 지체는 아주 높고 대역폭은 상대적으로 낮다. 예를 들어서 SMP 지체는 일반적으로 몇 마이크로초(역자주: 100만분의 1초)를 넘지 않지만 클러스터의 경우 일반적으로 수백 내지 수천 마이크로 초가 걸린다. SMP 통신 대역폭은 대개 100MBytes/sec이 넘는다; 비록 가장 빠른 네트웍 하드웨어(예, "기가비트 이더넷")가 이에 필적할 속도를 제공하긴 하지만 가장 일반적으로 사용되는 네트웍은 이보다 10에서 1000배 정도 더 느리다.

네트웍 하드웨어의 성능은 *고립된 클러스터 네트웍*에 충분할 만큼 낮다(The performance of network hardware is poor enough as an isolated cluster network). 네트웍이 다른 트래픽으로부터 고립되어 있지 않다면, 클러스터로 디자인된 시스템보다 "네트웍으로 묶은 기계들"을 사용한 경우가 더 많기 때문에 성능은 아주 악화될 수 있다.

- 클러스터를 단일 시스템으로 취급하는 소프트웨어 지원은 거의 없다. 예를 들어서 ps 명령은 단일 리눅스 시스템에 실행 중인 프로세스들에 대해서만 보고할 뿐 리눅스 시스템들로 만들어진 클러스터 전체에서 실행 중인 모든 프로세스들에 대해서 보고하지 않는다.

그래서, 클러스터는 굉장한 잠재력을 제공하지만 이 잠재력이 대부분의 어플리케이션들에 대해서 획득되기에는 아주 어려울 수 있다. 이런 환경에 적합한 프로그램들에 대해서 좋은 성능을 획득하도록 하는 많은 소프트웨어 지원이 있다는 것과 좋은 성능을 획득할 수 있는 프로그램들의 범위를 넓힐 수 있도록 특별히 설계된 네트웍들도 있다는 것은 좋은 소식이다.

3.2 네트웍 하드웨어(Network Hardware)

컴퓨터 네트워킹은 급증하고 있다... 이미 이것을 알고 있을 것이다. 네트워킹 기술과 제품들의 계속-증가하는 영역은 개발이 진행 중에 있고 대부분은 기계들(예, 각각 리눅스를 돌리는 피씨들)의 그룹에 병렬-처리 클러스터를 만드는 데 사용될 수 있는 형태로 사용 가능하다.

불행하게도 어떤 네트웍 기술도 모든 문제들을 훌륭하게 풀지 못한다; 사실 접근, 비용, 성능의 범위는 얼른 봐서 믿기 힘들다. 예를 들어서 표준 상업적으로-가능한 하드웨어를 사용해서 네트웍으로 기계들을 묶는 데 드는 비용은 기계 당 적게는 5달러에서 많게는 4000달러까지 이른다. 제조자가 말하는 대역폭(delivered bandwidth)와 지체 시간 각각은 크기의 네가지 등급(four orders of magnitude)에 따라 변한다.

특정 네트웍에 대해서 배우려고 하기 이전에 이런 것들은 바람처럼 쉽게 변한다는 것을 인식하는 것이 중요하다(리눅스 네트워킹 뉴스들에 대해서는 <http://www.uk.linux.org/NetNews.html>을 참조). 그리고 어떤 네트웍들에 대하 정확한 데이터를 얻는 것은 아주 어렵다. 특별히 확실하지 않는 곳에 저자는 물음표(*?*)를 놓았다. 이 주제를 연구하는 데 많은 시간을 썼지만 나는 내 요약(이 문서)이 잘못 투성이고 많은 중요한 것들을 빼먹었다는 것을 확신한다. 교정이나 추가해야 하는 것을 갖고 있다면 pplinux@ecn.purdue.edu로 이메일을 보내주기 바란다.

<http://web.syr.edu/~jmwobus/comfaqs/lan-technology.html>에 있는 LAN Technology Scorecard와 같은 요약들은 많은 서로 다른 타입들의 네트웍과 LAN 표준들에 대한 특성들을 보여준다. 그러나 이 하우투에 있는 요약은 대부분 리눅스 클러스터를 만드는 데 관련이 있는 네트웍 특성들에 대해서 촛점을 맞췄다. 각 네트웍을 논의하는 섹션은 짧은 특성 리스트로 시작한다. 다음은 이런 엔트리들이 의미하는 바를 정의한다.

리눅스 지원(Linux support):

(이것에 대한) 답변이 *no*라면, 그 의미는 분명하다. 다른 대답들은 네트웍을 억세스하는 데 사용되는 기본 프로그램 인터페이스를 설명하려고 할 것이다. 대부분의 네트웍 하드웨어는, 전형적으로 TCP/UDP 통신을 지원하는, 커널 드라이버를 통해서 인터페이스된다. 어떤 다른 네트웍들은 커널을 거치지 않고서 지체 시간을 좀 더 줄이기 위해서 좀 더 직접적인 인터페이스들(예, 라이브러리)을 사용하기도 한다.

몇년전 OS 호출을 통해서 부동 소숫점 유니트를 억세스 하는 것이 완전히 훌륭한 것으로 생각되어졌다. 그러나 이제 그것은 분명 우스운 것이다; 내 의견으로는 병렬 프로그램을 실행하는 프로세서들 간 각 통신이 OS 호출을 요구한다는 것은 어색한 것이다. 문제는 컴퓨터들이 아직도 이런 통신 메카니즘들을 통합하지 못했다는 것이다. 그래서 비-커널 접근은 이식성 문제들을 가지는 경향이 있다. 여러분은 가까운 미래에 이런 것에 대해서 좀 더 많은 것을 듣게 될 것이다. 대개 새로운 **Virtual Interface (VI) Architecture** <http://www.viarch.org/>의 형태로 듣게 될 것인 데 이것은 일반적인 OS 호출 계층을 피하는 대부분의 네트웍 인터페이스 작업들에 대한 표준화된 방법이다. VI 표

준은 Compaq, Intel, 그리고 Microsoft에 의해서 지원받고 있으며 다음 몇 년 안에 SAN(시스템 영역 네트웍) 디자인에 대한 강한 충격이 될 것임에 틀림없다.

최대 대역폭(Maximum bandwidth):

이것은 모든 사람이 신경쓰는 숫자이다. 나는 일반적으로 이론적으로 최선인 경우의 수치를 사용했다; 여러분의 마일리지는 변할 *것이다*.

최소 지체(Minimum latency):

내 의견으로는, 이것은 모든 사람들이 대역폭보다 더 신경써야 할 수치이다. 다시 나는 비현실적인 최선인 경우(base-case) 수치를 사용했지만 적어도 이 수치들은 하드웨어와 소프트웨어 모두를 포함하는 모든 지체 소스들을 포함한다. 대부분의 경우 네트웍 지체는 몇 마이크로초이다; 수치가 크면 클수록 하드웨어와 소프트웨어 인터페이스들의 계층들이 비효율적이다는 것을 반영한다.

구입 방법(Available as):

단순하게 말해서, 이것은 이 타입의 네트웍 하드웨어를 갖출 수 있는 방법을 설명한다. 상품들은 주요 구분 인자로 가격을 가지면서, 많은 벤더들에 의해서 살 수 있다. 다수-벤더에 의한 것들은 하나의 경쟁적인 벤더보다 좀 더 사기 쉽지만 이들은 중요한 차이와 잠재적인 호환성(interoperability) 문제들이 있다. 단일-벤더 네트웍은 공급자의 손에 완전히 종속된다(그러나 그들은 친철할 수도 있다). 퍼블릭 도메인 디자인이란 그것을 여러분에게 팔 사람을 찾지 못하더라도 부품들을 사서 그것을 만들 수 있다는 것을 말한다. 연구 프로토타입들은 말 그대로이다; 그들은 일반적으로 일반적으로 외부 사용자들에게 준비된 것이 아니거나 그들이 살 수 있는 것이 아니다.

사용된 인터페이스 포트/버스(Interface port/bus used):

이 네트웍을 어떻게 접속(hook-up)할 것인가? 현재 가장 높은 성능과 가장 일반적인 것은 PCI 버스 인터페이스 카드이다. EISA, VESA 로컬 버스(VL 버스), 그리고 ISA 버스 카드들도 있다. ISA는 맨처음에 나온 것이고 아직도 낮은-성능의 카드들에 대해서 많이 사용되는 것이다. EISA는 많은 PCI 기계들에서 두번째 버스로 사용되고 있어서 몇가지 카드들이 있다. 오늘날 VL 물건들을 많이 볼 수 없을 것이다(비록 `<http://www.vesa.org/>`가 의견을 달리하지만 말이다).

물론 여러분의 피씨의 케이스를 한번도 열어보지 않고 사용할 수 있는 어떤 인터페이스는 작은 매력 이상 가진다. IrDA와 USB 인터페이스들은 계속 빈번하게 나타나고 있다. 표준 패러럴 포트(SPP)는 프린터를 붙이는 데 사용되지만 ISA 버스의 외부 확장으로써 많이 사용되어 왔다; 이 새로운 기능은 EPP와 ECP 개선을 규정한 IEEE 1284 표준에 의해서 증진되었다. 또한 오래되고 신뢰성 있지만 느린 RS232 시리얼 포트가 있다. 나는 VGA 비디오 커넥터, 키보드, 마우스, 또는 게임 포트들을 사용해서 기계들을 연결하는 것을 알지 못한다... 그래서 여기에 없다.

네트웍 구조(Network structure):

버스는 구리 선이거나 구리 선들의 모임이거나 광섬유이다. 허브는 여기에 꼽혀 있는 서로 다른 구리 선/광섬유들을 연결하는 방법을 알고 있는 작은 박스이다; 스위칭 허브(switched hub)는 다수 커넥션들이 동시에 데이터를 전송하도록 하는 허브이다.

기계당 비용(Cost per machine connected):

여기서 이런 수치들을 사용하는 방법을 말한다. 네트웍 커넥션을 세지 않고서 클러스터의 한 노드에 쓸려고 피씨를 사는 데 2000달러가 들었다고 가정해보자. 패스트 이더넷(Fast Ethernet)을 더하는 것은 노드당 약 2400달러가 든다; Myrinet을 대신 더하는 데에는 약 3800달러가 든다. 2만달러가 있다면 Fast Ethernet에 연결된 8개의 기계나 Myrinet에 연결돈 5개의 기계를 가질 수 있다는 것을 의미한다. 멀티 네트웍을 가지는 것도 아주 의미있을 수 있다; 예, $20,000로 패스트 이더넷과 TTL_PAPERS 둘다 가지는 8개 기계들을 구매할 수 있다. 어플리케이션을 가장 빨리 실행할 클러스터를 만들 가능성이 가장 높은 네트웍이나 네트웍 집합을 선택하자.

이것을 읽는 현재 이런 수치들은 틀릴 수도 있다... 제기럴 그들은 아마도 이미 틀렸을 것이다. 또한 양이 줄어들거나 특별 판매(special deal)등이 있을 수 있다. 그러나 여기서 언급된 가격들때문에 여러분이 아주 부적절한 선택을 하게끔 하기에 충분한 잘못된 것일 가능성이 적다. 여러분의 어플리케이

션이 네트웍의 특별한 속성을 요구하거나 클러스터링된 피씨들이 상대적으로 비싼 것일 때에만 비싼 네트웍이 의미가 있다는 것을 아는 데는 박사 학위(비록 나는 하나 가지고 있지만 ;-)가 필요 없다.

내 의견을 들었으므로 쇼와 함께 다음에....(Now that you have the disclaimers, on with the show....)

3.2.1 아크넷(ArcNet)

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *2.5 Mb/s*
- 최소 지체: *1,000 microseconds?*
- 구입 방법: *멀티-벤터 하드웨어*
- 사용된 인터페이스 포트/버스: *ISA*
- 네트웍 구조: *스위치되지 않는(unswitched) 허브나 버스(논리적인 링)*
- 기계당 비용: *$200*

ARCNET은 주로 내장 실시간 제어 시스템들에서 사용되기 위해서 고안된 지역 네트웍(LAN)이다. 이더넷과 비슷하게 네트웍은 물리적으로 버스에 붙인 탭이나 하나 이상의 허브들로 조직된다. 그러나 이더넷과 다르게 이것은 네트웍을 논리적으로 링으로 구축하는 토큰-기반 프로토콜을 사용한다. 패킷 헤더는 작다(3 또는 4바이트). 그리고 메시지들은 단일 바이트 데이터만큼 작게 전달될 수 있다. 그래서, ARCNET은 제한된 지연 등을 가지면서, 이더넷보다 좀 더 일관된 성능을 가진다. 불행하게도 이것은 이더넷보다 더 느리고 덜 유명하다. 그러면서도 더 비싸다. `<http://www.arcnet.com/>`에 있는 ARCNET Trade Association로부터 더 자세한 정보를 얻을 수 있다.

3.2.2 ATM

- 리눅스 지원: *커널 드라이버, AAL* 라이브러리.*
- 최대 대역폭: *155 Mb/s* (곧, *1,200 Mb/s*)
- 최소 지체: *120 microseconds*
- 구매 방법: *멀티-벤더 하드웨어*
- 사용된 인터페이스 포트/버스: *PCI*
- 네트웍 구조: *스위치 허브*
- 기계 당 비용: *$3,000*

지난 몇년 동안 혼수 상태에 있지 않았다면 아마 ATM(비동기 전송 모드)가 어떻게 미래가.. 글쎄, 일종의 미래가 될 수 있는지에 대해서 많이 들었을 것이다. ATM은 HiPPI보다 더 싸고 패스트 이더넷보다 더 빠르며 전화 회사들이 제공하는 거리만큼 긴 거리에도 사용될 수 있다. ATM 네트웍 프로토콜은 또한 더 낮은-오버헤드 소프트웨어 인터페이스를 제공하도록, 그리고 작은 메시지들과 실시간 통신(예, 디지털 오디오와 비디오)을 좀 더 효과적으로 관리하도록 고안되었다. 이것은 또한 리눅스가 현재 지원하는 가장-높은 대역폭 네트웍들 중의 하나이다. 나쁜 소식은 ATM은 싸지 않고 벤더들 간에 호환성 문제가 아직 있다는 것이다. 리눅스 ATM 개발에 대한 개관은 `<http://lrcwww.epfl.ch/linux-atm/>`에서 찾을 수 있다.

3.2.3 CAPERS

- 리눅스 지원: *AFAPI 라이브러리*
- 최대 대역폭: *1.2 Mb/s*
- 최소 지체: *3 microseconds*
- 구매 방법: *상품 하드웨어*
- 사용된 인터페이스 포트/버스: *SPP*
- 네트워 구조: *2개의 기계들 간 케이블*
- 기계당 비용: *$2*

CAPERS(병렬 실행과 빠른 동기를 위한 케이블 어댑터; Cable Adapter for Parallel Execution and Rapid Synchronization)는, 전기 컴퓨터 엔지니어링의 퍼듀 대학 학교(Purdue University School of Electrical and Computer Engineering)에서의 PAPERS 프로젝트 `<http://garage.ecn.purdue.edu/~papers/>`의 부산물이다. 기본적으로 이것은 두 리눅스 피씨들에 대한 PAPERS 라이브러리를 구현하기 위해서, 일반 "LapLink" SPP-to-SPP 케이블을 사용하기 위해서 소프트웨어 프로토콜을 정의한다. 아이디어는 깍이지 않지만 가격을 더 낮출 수는 없다(역자주: 그만큼 싸다?). 시스템 보안을 개선하기 위해서 TTL_PAPERS와 마찬가지로 권고되는 마이너 커널 패치가 있다. 그러나 반드시 필요한 것은 아니다: `<http://garage.ecn.purdue.edu/~papers/giveioperm.html>`.

3.2.4 이더넷(Ethernet)

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *10 Mb/s*
- 최소 지체: *100 마이크로초*
- 구매 방법: *상품 하드웨어*
- 사용된 인터페이스 포트/버스: *PCI*
- 네트워 구조: *스위치 또는 스위치 없는 허브, 또는 허브 없는 버스*
- 기계당 비용: *$100*(허브 없는 경우 *$50*)

몇년동안 10 Mbits/s 이더넷은 표준 네트워 기술이 되었다. 좋은 이더넷 인터페이스 카드들은 $50 이하로 살 수 있다. 꽤 많은 피씨들이 이제는 마더보드에 이더넷 컨트롤러를 가지고 있다. 가볍게-사용되는 네트웍에 대해서 이더넷 연결은 허브 없는 멀티-탭 버스로 조직될 수 있다; 그런 설정은 최소의 비용으로 200개까지의 기계들을 묶을 수 있다. 그러나 병렬 처리에는 적절하지 않다. 더비 허브(unswitched hub)를 더하는 것은 실제 성능 향상에 도움이 되지 않는다. 그러나 동시 연결들에 대해서 전체 대역폭을 제공할 수 있는 스위치 허브(switched hub)들은 포트당 단지 약 $100만 든다. 리눅스는 놀라울 정도로 많은 이더넷 인터페이스들을 지원하지만 인터페이스 하드웨어의 변종들은 심각한 성능 차이를 부를 수 있다는 것을 기억하는 것이 중요하다. 어떤 것들이 지원되는지와 그들이 잘 작동하는지에 대해서 하드웨어 호환성 하우투(Hardware Compatibility HOWTO)를 보라; 그리고 다음을 보자 `<http://cesdis1.gsfc.nasa.gov/linux/drivers/>`.

성능을 향상하는 흥미로운 방법은 NASA CESDIS에서 수행된 비오울프(Beowulf) 프로젝트 `<http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html>`에서 수행된 16-기계 리눅스 클러스터에 의해서 제안되었다. 많은 이더넷 카드 드라이버의 저자인 Donald Becker는 각자 다른 것을 shadow하는(즉, 동일한 네트웍 주소들을 공유하는) 다수의 이더넷 네트웍을 통해서 로드 공유(load sharing)하는 지원을 개발하였

다. 이 로드 공유는 표준 리눅스 배포판에 내장되게 되었고 소켓 작업 레벨 아래에 보이지 않게 내장되게 되었다. 허브 비용이 중요하기 때문에 각 기계가 허브 없는 또는 더미 허브를 가진 두 개 이상의 이더넷 네트웍 (역자주: 카드)에 연결하는 것은 성능을 개선하기 위한 비용-효과적인 방법이 될 수 있다. 사실 한 기계가 네트웍 성능 병목에 걸린 상황에서 shadow 네트웍을 사용하는 로드 공유는 단일 스위치 허브 네트웍을 사용하는 것보다 훨씬 더 좋다.

3.2.5 이더넷(패스트 이더넷, Fast Ethernet)

- 리눅스 지원: *커널 드라이버.*
- 최대 대역폭: *100 Mb/s*
- 최소 지체: *80 마이크로 초*
- 구매 방법: *상품 하드웨어*
- 사용된 인터페이스 포트/버스: *PCI*
- 네트웍 구조: *스위치 또는 더미 허브*
- 기계당 비용: *$400?*

비록 그들을 "패스트 이더넷"으로 부르는 몇가지 다른 기술들이 실제 존재하지만 이 용어는 대개 옛날 "10 BaseT" 100 Mbits/s 장비와 케이블들과 다소 호환되는 허브-기반 100Mbits/s 이더넷을 가리킨다. 기대하는 것처럼 이더넷으로 불리는 것들은 어떤 것이나 일반적으로 용량으로 가격이 매겨지고 이런 인터페이스들은 일반적으로 155 Mbits/s ATM 카드들의 가격에 자그마한 조각에 지나지 않는다. 일단의 기계들이 단일 100 Mbits/s "버스" (더미 허브를 사용해서)의 대역폭을 나눠 쓰도록 하는 것은 각 기계의 연결에 풀로 10 Mbits/s를 제공할 수 있는 스위치 허브를 가지고 10 Mbits/s 이더넷을 사용하는 것보다 좋지 않을 수 있다는 함정(단점)이 있다.

각 기계에게 동시에 100 Mbits/s를 제공할 수 있는 스위치 허브는 비싸지만 가격이 매일 떨어지고 있고 이런 스위치들은 더미 허브보다 훨씬 더 높은 전체 네트웍 대역폭을 만든다. ATM 스위치들을 비싸게 만드는 요인은 그들이 각 ATM 셀(상대적으로 작은)들에 대해서 반드시 스위치해야 한다는 점이다; 어떤 패스트 이더넷 스위치들은 스위치를 지날 때 작은 지체를 가질 수 있는 기술들을 사용함으로써 기대되는 더 낮은 스위칭 주기의 이점을 이용한다. 그러나 스위치 패스를 변경하는 데 몇 밀리초(역자주: 마이크로 초가 아니다)가 걸린다... 그래서 라우팅 패턴이 자주 변한다면 이런 스위치는 피하는 것이 좋다. 여러 카드들과 드라이버들에 대해서는 `<http://cesdis1.gsfc.nasa.gov/linux/drivers/>`를 보라.

또한 이더넷에서 설명한 것처럼 NASA에서 이뤄진 Beowulf 프로젝트 `<http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html>`가 멀티 패스트 이더넷을 통해서 로드 공유함으로써 성능을 개선한 지원을 개발해오고 있다니 참고 바란다.

3.2.6 이더넷(기가비트 이더넷,Gigabit Ethernet)

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *1,000 Mb/s*
- 최소 지체: *300 마이크로 초?*
- 구매 방법: *멀티-벤더 하드웨어*
- 사용된 인터페이스 포트/버스: *PCI*
- 네트웍 구조: *스위치 허브 또는 FDR*
- 기계당 비용: *$2,500?*

기가비트 이더넷(Gigabit Ethernet) `<http://www.gigabitethernet.org/>`이 이더넷으로 불리는 좋은 기술적 이유를 가진다고 확신하지 못한다. 그러나 이것이 싸고 큰 시장이 있고 IP를 지원하는 컴퓨터 네트웍 기술을 갖고 있다는 것을 이름이 정확하게 의미하는 것은 아니다. 그러나 현재 가격은 Gb/s 하드웨어가 아직 만들기에 까다로운 것이라는 사실을 반영한다.

다른 이더넷 기술들과는 다르게 기가비트 이더넷은 좀 더 믿을 수 있는 네트웍을 만드는 흐름 제어의 레벨을 제공한다. FDR, 즉 풀-듀플렉스 리피터(Full-Duplex Repeater)는 성능을 향상시키기 위해서 버퍼링과 지역화된 흐름 제어를 사용하면서, 단순하게 라인들을 멀티플렉스한다. 대부분의 스위치 허브들은 현존하는 기가비트-가능 광섬유 스위치(gigabit-capable switch fabrics)에 대한 새로운 인터페이스 모듈들로써 구축되고 있다. 스위치/FDR 제품들은 적어도 다음과 같은 사이트들에서 구매될 수 있거나 발표되고 있다. `<http://www.acacianet.com/>`, `<http://www.baynetworks.com/>`, `<http://www.cabletron.com/>`, `<http://www.networks.digital.com/>`, `<http://www.extremenetworks.com/>`, `<http://www.foundrynet.com/>`, `<http://www.gigalabs.com/>`, `<http://www.packetengines.com/>`. `<http://www.plaintree.com/>`, `<http://www.prominet.com/>`, `<http://www.sun.com/>`, and `<http://www.xlnt.com/>`.

리눅스 드라이버 `<http://cesdis.gsfc.nasa.gov/linux/drivers/yellowfin.html>`가 존재하며 Packet Engines "Yellowfin" G-NIC에 대해서는 `<http://www.packetengines.com/>`. 리눅스에서 한 초기 테스트는 가장 좋은 100 Mb/s 패스트 이더넷으로 획득될 수 있는 것보다 약 2.5배 높은 대역폭을 얻었다; 기가비트 네트웍의 경우PCI 버스 사용을 조심스럽게 튜닝하는 것이 중요한 인자이다. 의심할 바 없이 드라이버 개선과 다른 NIC들에 대한 리눅스 드라이버 지원이 계속 이어질 것이다.

3.2.7 FC (광섬유 채널, Fibre Channel)

- 리눅스 지원: *no*
- 최대 대역폭: *1,062 Mb/s*
- 최소 지체: *?*
- 구매 경로: *멀티-벤더 하드웨어*
- 인터페이스 포트/버스: *PCI?*
- 네트웍 구조: *?*
- 기계 당 가격: *?*

FC(광섬유 채널)의 목적은 높은-성능 블럭 I/O(2,048 바이트 데이터 하중을 실어나르는 FC 프레임)를 제공하는 것이다. 특별히 컴퓨터를 통해서가 아니라 FC에 직접 연결될 수 있는 다른 저장 장치와 디스크들을 공유하기 위해서 말이다. 대역폭-별로 FC는 133에서 1,062 Mbits/s 사이 어디에서나 실행되면서 상대적으로 빠르다고 한다. FC가 high-end SCSI를 대체할 만큼 유명해진다면 이것은 값싼 기술이 될 것이다; 지금은 값싼 기술이 아니며 리눅스에 의해서 지원되지 않는다. FC 레퍼런스들에 대한 좋은 콜랙션은 `<http://www.amdahl.com/ext/CARP/FCA/FCA.html>`에서 Fibre Channel Association에 의해서 관리되고 있다.

3.2.8 파이어와이어(FireWire, IEEE 1394)

- 리눅스 지원: *no*
- 최대 대역폭: *196.608 Mb/s* (곧, *393.216 Mb/s*)
- 최소 지체: *?*
- 구매 방법: *멀티벤더 하드웨어*

- 인터페이스 포트/버스: *PCI*
- 네트웍 구조: *사이클 없는 랜덤(자가-설정)*
- 기계당 비용: *$600*

FireWire, <http://www.firewire.org/>, IEEE 1394-1995 표준인 이것은 고객 전자장비(electronics)를 위한 저비용 고속 디지탈 네트웍을 위해서 고안된 것이다. 진열장 어플리케이션은 DV 디지털 비디오 캠코더를 컴퓨터에 연결하는 것이지만 FireWire는 SCSI를 대체하는 것부터 여러분의 가정극장(home theater)의 각 컴포넌트들을 상호연결하는 것까지의 어플리케이션들을 위하여 사용되도록 의도되었다. 이것은 원(cycle)을 만들지 않는 버스들과 브리지들을 사용하는 임의의 토폴로지에서 연결된 64K 장비들까지 허락하고 컴포넌트들이 더해지거나 제거될 때 자동으로 설정을 검출한다. 짧고(4 바이트 "quadlet") 낮은 지체 시간을 가지는 메시지들이 ATM-like한 동기(isochronous) 전송(멀티미이더 메시지들의 동기를 맞추는 데 사용된다)과 함께 지원된다. 아답텍(Adaptec)은 단일 PCI 인터페이스 카드에 63개까지 장치들을 허락하는 FireWire 제품들을 가지고 있고 <http://www.adaptec.com/serialio/>에 FireWire에 대한 좋은 일반적인 정보를 담고 있다.

비록 FireWire가 사용 가능한 가장 높은 대역폭 네트웍이 될 수는 없겠지만 고객-레벨 마켓(가격을 아주 낮게 만들)과 낮은 지체 시간 지원은 이것을 몇년 이내에 리눅스 피씨 클러스터 메시지-전달 네트웍 기술들 중의 하나로 만들 것이다.

3.2.9 HiPPI과 시러얼 HiPPI

- 리눅스 지원: *no*
- 최대 대역폭: *1,600 Mb/s* (시리얼은 *1,200 Mb/s*)
- 최소 지체: *?*
- 구입 방법: *멀티-벤더 하드웨어*
- 사용된 인터페이스 포트/버스: *EISA, PCI*
- 네트웍 구조: *스위치 허브*
- 기계당 비용: *$3,500* (시리얼은 *$4,500*)

HiPPI (High Performance Parallel Interface)는 원래 슈퍼 컴퓨터와 다른 기계(슈퍼컴, 프레임 버퍼, 디스크 어레이 등) 간에 대량의 데이터 셋들의 전송을 위해서 아주 높은 대역폭을 제공하도록 의도되었으며 슈퍼컴에 대한 지배적인 ㅍ준이 되었다. 비록 이것은 모순 어법(oxymoron)이긴 하지만 **시리얼 HiPPI** 도 또한 32-비트 폭 표준(패러럴) HiPPI 케이블 대신에 광섬유 케이블을 전형적으로 사용함으로써, 유명해질 것이다. 지나 몇 년동안 HiPPI 크로스바 스위치들은 일반적인 것이 되었으며 가격들은 급격하게 떨어졌다; 불행하게도 시리얼 HiPPI는 아직 비싸고 이것이 바로 PCI 버스 인터페이스 카드들은 일반적으로 지원하는 것이다. 더 나쁜 것은 리눅스는 아직 HiPPI를 지원하지 못한다는 것이다. HiPPI의 좋은 개관은 CERN에 의해서 <http://www.cern.ch/HSI/hippi/>에서 관리되고 있다; 그들은 또한 <http://www.cern.ch/HSI/hippi/procintf/manufact.htm>에 HiPPI 벤더들의 다소 기다란 리스트를 관리한다.

3.2.10 IrDA (적외선 데이터 연합; Infrared Data Association)

- 리눅스 지원: *no?*
- 최대 대역폭: *1.15 Mb/s와 4 Mb/s*
- 최소 지체: *?*
- 구매 방법: *멀티-벤더 하드웨어*

- 인터페이스/버스: *IrDA*
- 네트웍 구조: *짧은 거리 무선(thin air) ;-)*
- 기계당 비용: *$0*

IrDA(적외선 데이터 연합, `http://www.irda.org/`)는 많은 랩톱 피시들의 측면에 있는 작은 적외선 장치이다. 이 인터페이스를 사용해서 두 개 이상의 기계들을 커넥트하는 것은 타고 날 때부터 어렵다. 그래서 클러스터링에 사용될 가능성이 낮다. Don Becker가 IrDA에 몇가지 예비 작업을 했었다.

3.2.11 Myrinet

- 리눅스 지원: *라이브러리*
- 최대 대역폭: *1,280 Mb/s*
- 최소 지체: *9 마이크로 초*
- 구매 방법: *단일-벤더 하드웨어*
- 인터페이스 포트/버스: *PCI*
- 네트웍 구조: *스위치 허브*
- 기계당 가격: *$1,800*

Myrinet `<http://www.myri.com/>`은 "시스템 영역 네트웍" (System Area Network, SAN)으로 사용될 수 있도록 고안된 근거리 영역 네트웍(LAN)이다. 즉 병렬 시스템으로 연결된 기계들이 가득찬 네트웍 케비넷. LAN과 SAN 버전들은 서로 다른 물리적 매체를 사용하고 다소 다른 특성들을 가진다; 일반적으로 SAN 버전은 클러스터 안에서 사용될 것이다.

Myrient은 구조적으로 아주 전통적인 것이지만 특별히 잘-구현된 것이라는 평판을 듣고 있다. 리눅스를 위한 드라이버는 성능이 좋다는 말을 듣는다. 비록 호스트 컴퓨터에 대한 서로 다른 PCI 버스 구현물들에 대해서 여러 커다란 성능 변화들이 리포트된 바가 있었지만 말이다.

현재 Myrinet은 너무 심각하게 "예산을 위협하는" 것이 아닌 클러스터 그룹의 선호되는 네트웍임에 틀림없다. 리눅스 피씨에 대한 생각이 최소 256 MB RAM과 SCSI RAID를 가진 Pentium Pro나 Pentium II라면 Myrinet의 가격은 꽤 합리적인 것이 된다. 그러나 좀 더 일반적인 피씨 설정을 사용한다면 여러분의 선택이 Myrinet에 연결된 N 기계들이나 멀티 패스트 이너넷으로 묶인 $2N$과 TTL_PAPERS 사이에 있다는 것을 알게 될 것이다. 여러분의 예산이 얼마나 되는가와 여러분이 신경쓰는 컴퓨터의 사양이 어떤 것인가에 따라 좌지 우지 된다.

3.2.12 파라스테이션(Parastation)

- 리눅스 지원: *HAL 또는 소켓 라이브러리*
- 최대 대역폭: *125 Mb/s*
- 최소 지체: *2 microseconds*
- 구입 방법: *단일-벤더 하드웨어*
- 사용된 인터페이스/버스: *PCI*
- 네트웍 구조: *허브없는 망(mesh)*
- 기계당 비용: *> $1,000*

Karlsruhe 대학교 정보공학과(Department of Informatics)의 파라스테이션(Parastation) 프로젝트 `<http://wwwipd.ira.uka.de/parastation>`는 PVM-호환 커스텀 저-지체 네트웍을 구축 중이다. 그들은 맨처음 커스텀 EISA 인터페이스와 BSD UNIX를 실행하는 피씨들을 사용한 맨처음 구축된 두개의 프로세서 ParaPC 프로토타입을 만들었으며 그 다음 DEC Alpha들을 사용한 좀 더 큰 클러스터들을 만들었다. PCI 카드들은 Hitex이라고 불리는 회사와 공조해서 만들어졌다(`<http://www.hitex.com:80/parastation/>` 참조). 파라스테이션 하드웨어는 빠르고 신뢰성 있는 메시지 전송과 단순한 관문(barrier) 동기화를 구현한 것이다.

3.2.13 PLIP

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *1.2 Mb/s*
- 최소 지체: *1,000 microseconds?*
- 구입 방법: *상품 하드웨어*
- 사용된 인터페이스/버스: *SPP*
- 네트웍 구조: *2 기계 간 케이블*
- 기계당 비용: *$2*

"LapLink" 케이블의 비용으로 PLIP(Parallel Line Interface Protocol)는 표준 소켓-기반 소프트웨어를 사용하여 표준 패러럴 포트들을 통해서 두 리눅스 기계들이 통신할 수 있도록 한다. 대역폭, 지체, 그리고 측정 가능성(scalability)의 측면에서 보면 이것은 아주 중요한 네트웍 기술이 아니다. 그러나 거의 영에 가까운 가격과 소프트웨어 호환성이 유용하다. 드라이버는 표준 리눅스 커널 배포판의 일부이다.

3.2.14 SCI

- 리눅스 지원: *no*
- 최대 대역폭: *4,000 Mb/s*
- 최소 지체: *2.7 microseconds*
- 구입 방법: *멀티-벤더 하드웨어*
- 사용된 인터페이스/버스: *특정회사(proprietary) PCI*
- 네트웍 구조: *?*
- 기계당 비용: *> $1,000*

SCI (Scalable Coherent Interconnect, ANSI/IEEE 1596-1992)의 목적은 기본적으로 많은 수의 기계들에 걸쳐 근접(coherent) 공유 메모리 공유를 지원할 수 있는 고성능 메카니즘과 다양한 타입의 메시지 전송을 제공하는 것이다. SCI의 고안된 대역폭과 지체는 대부분의 다른 네트웍 기술들과 비교해서 둘 다 "놀라운 것"이라고 말해도 괜찮다. 단점은 SCI가 싼 제품 유니트로써 널리 사용가능하지 않다는 것과 리눅스 지원이 아직 없다는 것이다.

SCI는 주로 HP/Convex Exemplar SPP와 Sequent NUMA-Q 2000(`<http://www.sequent.com/>`)와 같은 지역적으로-공유된 물리적으로-배포된 메모리 기계들(logically-shared physically-distributed memory machines)에 대한 다양한 고유 디자인 안에서 사용되었다. 그러나 Dolphin(`<http://www.sequent.com/>` 참조)로부터 SCI는 PCI 인터페이스 카드와 4-way 스위치(16 기계들까지 네개의 4-way 스위치들을 붙여서 연결될 수 있다)들이 그들의 CluStar 제품 라인으로써 사용가능하다. SCI 개관에 대한 좋은 링크들은 CERN 에 의해서 `<http://www.cern.ch/HSI/sci/sci.html>`에서 관리된다.

3.2.15 SCSI

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *5 Mb/s*에서 *20 Mb/s* 이상까지
- 최소 지체: *?*
- 구입 방법: *멀티-벤더 하드웨어*
- 사용된 인터페이스/버스: *PCI, EISA, ISA 카드*
- 네트웍 구조: *SCSI 장비들을 공유하는 기계간 버스*
- 기계당 비용: *?*

SCSI (Small Computer Systems Interconnect)는 기본적으로 디스크 드라이브들, CDROM들, 이미지 스캐너 등과 같은 데 사용되는 I/O 버스이다. 여기에는 분리된 세개의 표준들 SCSI-1, SCSI-2, 그리고 SCSI-3; Fast and Ultra speeds; 그리고 데이터 패스 너비로 8, 16, 또는 32비트(SCSI-3에서 언급된 FireWire 호환성과 함께) 가 있다. 이것은 아주 혼란스럽다. 그러나 우리는 모두 좋은 SCSI는 EIDE보다 더 빠르고 장비들을 좀 더 효율적으로 관리할 수 있다는 것을 안다.

많은 사람들이 깨닫지 못하는 것은 두 컴퓨터들이 단일 SCSI 버스를 공유하는 것이 아주 단순하다는 사실이다. 이런 설정 타입은 기계간 디스크 드라이브들을 공유하고 **장애 복구(fail-over)**를 구현-한 기계가 다른 기계가 실패했을 때 데이터베이스 요구를 떠맡도록 하게 함으로써-에 아주 유용하다. 현재 이것은 마이크로소프트의 PC 클러스터 제품 WolfPack에 의해서 지원되는 유일한 메카니즘이다. 그러나 좀 더 큰 시스템들로 확장될 수 없다는 것이 공유 SCSI를 일반적인 병렬 처리에 있어 덜 흥미롭게 만든다.

3.2.16 서버넷(ServerNet)

- 리눅스 지원: *no*
- 최대 대역폭: *400 Mb/s*
- 최소 지체: *3 microseconds*
- 구입 방법: *단일-벤더 하드웨어*
- 사용된 인터페이스/버스: *PCI*
- 네트웍 구조: *허브의 육각 트리/4면체 격자(hexagonal tree/tetrahedral lattice of hubs)*
- 기계당 비용: *?*

서버넷(ServerNet)은 Tandem, `<http://www.tandem.com>`의 고-성능 네트웍 하드웨어이다. 특별히 온라인 트랜잭션 처리(OLTP) 세계에서 탠덤은 고-신뢰도 시스템들의 선두로 알려져 있어서 그들의 네트웍이 고-성능 뿐만이 아니고 "높은 데이터 통합(integrity)과 신뢰도"까지 주장해도 놀라운 일이 아니다. ServerNet의 다른 흥미로운 면은 임의의 장치에서 다른 임의의 장치로 직접 데이터를 전송할 수 있다고 주장한다는 것이다; 3.5 (MPI) 섹션에서 설명된 MPI 리모트 메모리 억세스 메카니즘들에 의해서 제안된 것과 비슷한 원-사이드 스타일로, 프로세서들사이뿐만이 아니고 디스크 드라이브들 등 사이에서도 그렇다. 서버넷에 대한 마지막 한가지 코멘트: 단지 싱글 벤더만 존재하지만 그 벤더는 서버넷을 주요 표준으로, 잠재적으로 만들 수 있을 만큼 충분히 강력하다. Tandem은 Compaq 소유이다.

3.2.17 SHRIMP

- 리눅스 지원: *사용자-레벨 메모리 맵된(memory-mapped) 인터페이스*
- 최대 대역폭: *180 Mb/s*
- 최소 지체: *5 microseconds*
- 구입 방법: *연구 프로토타입*
- 사용된 인터페이스/버스: *EISA*
- 네트워 구조: *mesh backplane (인텔 Paragon와 유사)*
- 기계당 비용: *?*

프린스턴 대학교 컴퓨터 과학 학과에서 진행하고 있는 SHRIMP 프로젝트, `<http://www.CS.Princeton.EDU/shrimp/>`는 처리 노드로써 리눅스를 실행하는 피씨들을 사용한 병렬 컴퓨터를 구축하고 있다. 첫번째 SHRIMP(Scalable, High-Performance, Really Inexpensive Multi-Processor의 약자)는 커스텀 EISA 카드 인터페이스 위의 듀얼 포트를 가지는 RAM을 사용한 단순한 두-프로세서 프로토 타입이었다. 지금은 인텔 Paragon(`<http://www.ssd.intel.com/paragon.html>` 참조)에서 사용된 그물망(mesh) 라우팅 네트워과 기본적으로 동일한 "hub"에 연결하기 위해서 커스텀 인터페이스를 사용하는 좀 더 큰 설정들에 확장가능한 프로토타입이 존재한다. 오베헤드가 적은 "가상 메모리 맵된 통신" 하드웨어와 지원 소프트웨어를 개발하기 위한 상당히 많은 노력이 이루어졌다.

3.2.18 SLIP

- 리눅스 지원: *커널 드라이버*
- 최대 대역폭: *0.1 Mb/s*
- 최소 지체: *1,000 microseconds?*
- 구입 방법: *상품 하드웨어*
- 사용된 인터페이스/버스: *RS232C*
- 네트워 구조: *두 기계들 사이에 케이블 사용*
- 기계당 비용: *$2*

비록 SLIP(시리얼 라인 인터페이스 프로토콜;Serial Line Interface Protocol)은 성능 스펙트럼에서 낮은 쪽에 완전히 위치하고 있지만 SLIP(또는 CSLIP또는 PPP)는 두 기계들이 일반 RS232 시리얼 포트들을 통해서 소켓 통신을 수행할 수 있도록 한다. 또는 그들은 모뎀을 경유한 다이얼-업을 통해서 연결될 수도 있다. 어떤 경우에도 지체 시간은 높고 대역폭은 낮다. 그래서 SLIP은 다른 대안들이 전혀 불가능할 때만 사용되어야 할 것이다. 그러나 대부분의 피씨드이 두 개의 RS232 포트들을 갖고 있다는 것은 주목할만한 가치가 있다. 그래서 기계들을 선형 배열이나 링형으로 단순하게 연결해서 기계들 그룹을 네트워킹하는 것이 가능하다. EQL이라고 불리는 로드 쉐어링 소프트웨어도 존재한다.

3.2.19 TTL_PAPERS

- 리눅스 지원: *AFAPI 라이브러리*
- 최대 대역폭: *1.6 Mb/s*
- 최소 지체: *3 microseconds*

- 구입 방법: *퍼블릭-도메인 설계, 싱글-벤더 하드웨어*
- 사용된 인터페이스/버스: *SPP*
- 네트웍 구조: *허브들의 트리*
- 기계당 비용: *$100*

퍼듀 대학교의 전자 및 컴퓨터 엔지니어링 학교에서 수행 중인 PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) 프로젝트, `<http://garage.ecn.purdue.edu/~papers/>`는 병렬 슈퍼컴퓨터가 변조되지 않은 피씨들/워크스테이션들을 노드로 사용하여 구축될 수 있도록 하는, 크기 조절 가능하고 낮은-지체 시간을 가지며 집합 함수 통신 하드웨어와 소프트웨어를 구축 중에 있다.

두가지 개발 라인들을 대략 따르는 SPP(표준 패러럴 포트; Standard Parallel Port)를 통해서 피씨들/워크스테이션들을 연결한 PAPERS 하드웨어는 그 종류가 12가지가 넘는다. "PAPERS"라고 불리는 버전들은 적절한 기술이면 무엇이든지 사용해서 더 높은 성능을 목표로 한다; 현재 작업은 FPGA들과 지금 개발 중에 있는 높은 대역의 PCI 버스 인터페이스 설계들을 사용한다. 반면에 "TTL_PAPERS"이라고 불리는 버전들은 퍼듀 대학 바깥에서 쉽게 재생산될 수 있도록 디자인된 것이고 일반적인 TTL 로직을 사용해서 만들어질 수 있는 아주 단순한 공용 도메인 설계(public domain designs)이다. 이런 디자인 중 하나는 상용으로 만들어졌다. `<http://chelsea.ios.com:80/~hgdietz/sbm4.html>`

다른 대학들의 커스텀 하드웨어 디자인과 다르게 TTL_PAPERS 클러스터들은 USA에서 대한민국까지 많은 대학들에서 조립되어 왔다. 대역폭은 SPP 커넥션들에 의해서 심각하게 제한되어 있지만 가장 빠른 메시지-기반 시스템들은 그러한 집합 함수들(aggregate functions)에 대해서 필적할만한 성능을 제공할 수 없다. 그래서 PAPERS는 특별히 비디오 벽(video wall; 역자주: 전시장 등에서 볼 수 있는 대형 디스플레이 및 컴퓨터, 사운드 시스템의 조합)의 디스플레이를 동기화하는 데(비디오 벽에 대해서 앞으로 나올 Video Wall HOWTO에서 자세히 다룰 것이다), 고-대역 네트웍에 대한 스케줄링 억세스, 유전자 검색(genetic search)에서 전역 비교(global fitness)를 평가하는 것 등에서 탁월하다. 비록 PAPERS 클러스터들이 IBM PowerPC AIX, DEC Alpha OSF/1, 그리고 HP PA-RISC HP-UX 기계들을 사용해서 만들어졌지만 리눅스-기반 PC들이 가장 잘 지원되는 플랫폼이다.

TTL_PAPERS AFAPI를 사용하는 사용자 프로그램들은 리눅스에서, 각 억세스에 대해서 OS 호출없이 SPP 하드웨어 포트 레지스터들을 억세스한다. 이렇게 하기 위해서 AFAPI는 맨먼저 `iopl()`나 `ioperm()`를 사용해서 포트 퍼미션을 획득한다. 이런 호출들의 문제점은 둘 다 사용자 프로그램이 권한을 갖도록(역자주: 아무래도 루트 권한일 것 같다) 요구해서 잠재적인 보안 구멍을 만든다는 것이다. 솔루션은 선택적인 커널 패치, `<http://garage.ecn.purdue.edu/~papers/giveioperm.html>`이다. 이것은 권한있는 프로세스가 임의의 프로세스에 대한 포트 퍼미션을 제어하도록 한다.

3.2.20 USB (Universal Serial Bus)

- 리눅스 지원: *kernel driver*
- 최대 대역폭: *12 Mb/s*
- 최소 지체: *?*
- 구입 방법: *commodity hardware*
- 사용된 인터페이스/버스: *USB*
- 네트웍 구조: *bus*
- 기계당 비용: *$5?*

USB (Universal Serial Bus, `<http://www.usb.org/>`)는 키보드, 화상 회의 카메라 등 127개까지 주변기기들을 달 수 있고 핫-플러그(hot-pluggable) 가능한 일반 이더넷 수준의 속도를 내는 버스이다. 얼마

나 많은 컴퓨터들이 서로 USB를 사용해서 연결될 수 있는지는 실제 명확하지 않다. 어쨌든 USB 포트들은 지금 빠르게 RS232와 SPP와 같은 PC 마더보드의 표준이 되어가고 있다. 그러므로 여러분이 구매한 차기 PC의 뒤편에 USB 포트들이 숨어 있더라도 놀라지 말기 바란다. 리눅스 드라이버 개발은 `<http://peloncho.fis.ucm.es/~inaky/USB.html>`에서 논의되고 있다.

여러가지 점에서 USB는 거의 여러분이 현재 구매할 수 있는, 낮은-성능, 제로-비용(zero-cost)인 FireWire 버전이다.

3.2.21 WAPERS

- 리눅스 지원: *AFAPI library*
- 최대 대역폭: *0.4 Mb/s*
- 최소 지체: *3 microseconds*
- 구입 방법: *public-domain design*
- 사용된 인터페이스/버스: *SPP*
- 네트워 구조: *wiring pattern between 2-64 machines*
- 기계당 비용: *$5*

WAPERS (병렬 실행과 빠른 동기화를 위한 Wired-AND 아답터; Wired-AND Adapter for Parallel Execution and Rapid Synchronization)는 퍼듀 대학교의 전자 컴퓨터 공학 학교에서 수행 중인 PAPERS 프로젝트, `<http://garage.ecn.purdue.edu/~papers/>`의 부산물이다. 적절하게 구현된다면 SPP는 4-비트 폭 wired AND를 구현하기 위해서 기계들 간 서로 묶일 수 있는 4비트 오픈-콜렉터 출력을 가진다. 이 wired-AND는 전자공학적으로 다루기 어려운 것이고 이런 식으로 연결될 수 있는 기계들의 최대 개수는 포트의 아날로그 특성에 종속적이다(최대 수신 전류(sink current)와 휴식 레지스터(pull-up register) 값); 전형적으로 7개 내지 8개 기계들이 WAPERS로 네트워킹될 수 있다. 비록 비용과 지체시간이 아주 낮지만, 그래서 대역폭도 낮다; WAPERS는 클러스터에서 단일 네트워으로써가 아니라 집합 작업들에 대한 두번째 네트워으로써 훨씬 더 좋다. TTL_PAPERS와 함께, 시스템 보안을 높이기 위해서, 반드시 필요하지는 않지만 권고되는 마이너 커널 패치가 있다: `<http://garage.ecn.purdue.edu/~papers/giveioperm.html>`.

3.3 네트워 소프트웨어 인터페이스(Network Software Interface)

병렬 어플리케이션들을 지원하는 소프트웨어를 논의하기 이전에 네트워 하드웨어에 대한 로우-레벨 소프트웨어 인터페이스의 기본을 간단히 먼저 얘기하는 것이 유용하다. 실제로 3가지 기본 선택만 존재한다: 소켓, 장치 구동기(device drivers), 그리고 유저-레벨 라이브러리.

3.3.1 소켓

지금까지 가장 일반적인 로우-레벨 네트워 인터페이스는 소켓 인터페이스이다. 소켓은 지난 10년간 유닉스의 일부였고 대부분의 표준 네트워 하드웨어는 적어도 두가지 타입의 소켓 프로토콜들: UPD와 TCP를 지원하도록 설계된 것이다. 두 소켓 타입들은 한 기계에서 다른 것으로 임의 크기의 데이터 블럭을 전송할 수 있도록 하지만 몇가지 중요한 차이가 있다. 비록 성능은 네트워 트래픽에 따라서 훨씬 악화될 수 있지만 전형적으로 이 두가지는 약 1,000 마이크로 초 정도의 최소 지체를 만든다.

이런 소켓 타입들은 대부분의 이식 가능, 하이-레벨, 병렬 처리 소프트웨어에 대한 기본 네트워 소프트웨어 인터페이스이다; 예를 들어서 PVM은 UDP와 TCP를 혼합하여 사용하기 때문에 이 둘의 차이점을 아는 것은 성능을 튜닝하는 데 도움을 줄 것이다. 좀 더 나은 성능을 위해서 프로그램 안에서 직접 이런 메카니즘들을 사용할 수도 있다. 다으은 UDP와 TCP의 단순한 개관이다; 자세한 내용은 매뉴얼 페이지들과 좋은 네트워 프로그래밍 책을 보기 바란다.

UDP 프로토콜 (SOCK_DGRAM) **UDP**는 사용자 데이터그램 프로토콜(User Datagram Protocol)이지만 UDP의 속성을 신뢰할 수 없는 데이터그램 처리(Unreliable Datagram Processing)로 좀 더 쉽게 기억할 수 있을 것이다. 다른 말로 해서 UDP는 각 블럭이 개별 메시지로 전송되도록 허락하지만 메시지는 전송 중 유실될 수 있다. 사실 네트웍 트래픽에 종속적으로 UDP 메시지들은 유실될 수 있고 여러번 도착할 수 있거나 그들이 보내진 순서와 다른 순서로 도착할 수 있다. UDP 메시지의 전송자는 자동으로 받았다는 통지(acknowledgement)를 받지 않는다. 그래서 이런 문제들을 검출하고 보충하는 것은 사용자가 작성한 코드에 의존한다. 다행스럽게도 UDP는 메시지가 도착했다면 받은 메시지가 손상된 것이 아니고 완전한 것이라고(즉, UDP 메시지 조각만 받았다고) 보장한다.

UDP가 좋은 점은 가장 빠른 소켓 프로토콜이 되려고 한다는 것이다. 더 나아가 UDP는 "연결 없는(connectionless)" 것이다. 이것은 각 메시지가 기본적으로 모든 다른 것들과 독립이다는 것을 의미한다. 이것에 대한 좋은 비유는 편지이다; 여러분은 동일한 집주소로 여러 편지를 보낼 수 있지만 각각은 다른 것들과 독립이며 여러분이 편지를 보낼 수 있는 사람의 수에 대한 제한이 없다.

TCP 프로토콜(SOCK_STREAM) UDP와 다르게 **TCP**는 신뢰할 수 있고, 연결-기반인 프로토콜이다. 보내진 각 블럭은 메시지로 보이지 않고 겉보기에 전송자와 수신자 사이의 연결을 통해서 전송된 연속적인 바이트안에서 데이터 블럭으로 보인다. 이것은 UDP 메시징과 아주 다르다. 왜냐면 각 블럭은 단순하게 바이트 스트림의 일부이고 각 블럭을 바이트 스트림에서 추출하는 방법을 알아 내는 것은 사용자 코드에 종속적이기 때문이다; 메시지들을 분리하는 마킹은 없다. 더 나아가 네트웍 문제들에 대해서 연결은 좀 더 깨지기 쉬운 것이고 연결의 제한된 개수들만이 각 프로세스들에 대해서 동시에 존재할 수 있다. 이것은 신뢰할 수 있기 때문에 TCP는 일반적으로 UDP보다 좀 더 무거운 오버헤드를 가진다.

그러나 TCP에 관한 몇가지 즐거운 놀라운 것들이 존재한다. 다수 메시지들이 연결을 통해서 전달되었다면, 짧은 또는 짝이 맞지 않은 크기의 메시지들의 그룹에 대해서 UDP보다 더 나은 성능을 잠재적으로 내면서, TCP는 그것들을 버퍼 안에서 네트웍 하드웨어 패킷 크기에 더 잘 맞도록 묶을 수 있다는 것이 첫번째이다. 다른 보너스는 기계들 간에 신뢰할 수 있는 직접 물리적 링크들을 사용해서 구축된 네트웍은 TCP 연결을 쉽고 효율적으로 흉내낼 수 있다는 것이다. 예를 들어서 ParaStation의 "Socket Library" 인터페이스 소프트웨어의 경우 이렇게 되었다. 이 소프트웨어는 표준 TCP OS 호출들과 각 함수 이름에다 접두사 `PSS`를 붙이는 것만 다른 사용자-레벨 호출들을 사용한 TCP 문법(semantics)을 제공한다.

3.3.2 장치 구동기(Device Drivers)

네트웍에 데이터를 실제로 넣을 때가, 또는 네트웍으로부터 데이터를 끄집어 올 때가 오면 표준 유닉스 소프트웨어 인터페이스는 장치 구동기라고 불리는 유닉스 커널의 일부가 된다. UDP와 TCP는 단지 데이터만 전송하지 않고 그들은 또한 상당한 양의 소켓 관리의 오버헤드를 갖고 있다. 예를 들어서 어떤 것들은 다수의 TCP 커넥션들이 하나의 물리적 네트웍 인터페이스를 공유할 수 있다는 사실을 관리해야 한다. 이에 비해서 전용 네트웍 인터페이스에 대한 장치 구동기는 단지 몇가지 단순한 데이터 전송 함수들만 구현하면 된다. 이런 장치 드라ㅇ이버 함수들은 사용자 프로그램에 의해서, 적절한 장치를 확인하기 위해서 `open()`을 사용하고 오픈된 "파일"에 대해서 `read()`와 `write()`와 같은 시스템 호출을 사용함으로써, 호출될 수 있다. 그래서 각각의 그런 작업은 데이터 블럭을 시스템 호출의 오버헤드보다 더 작은 오버헤드로 전송할 수 있다. 이런 시스템 호출은 수십 마이크로 초가 걸린다.

리눅스에 대한 장치 구동기를 작성하는 것은 어렵지 않다... 장치 하드웨어가 작동하는 방법을 *정확하게* 알고 있다면 말이다. 이것이 작동하는 방법을 모른다면 추측하지 말라. 장치 드라이버를 디버깅하는 것은 즐겁지 않은 일이고 실수들은 하드웨어를 태워먹을 수 있다. 그러나 이것이 여러분을 그렇게 겁주는 것이 아니라면, 예를 들어서 전용 이더넷 카드를 더미로 그러나 일반적인 이더넷 프로토콜 오버헤드 없이 기계-대-기계 빠른 직접 연결로 사용하기 위해서, 장치 구동기를 작성하는 것은 가능한 일이 될 수 있다. 사실 초기 인텔 슈퍼컴퓨터들이 했던 것과 상당히 유사하다.... 좀 더 자세한 정보를 보고 싶다면 Device Driver HOWTO를 보라.

3.3.3 사용자-레벨 라이브러리(User-Level Libraries)

여러분이 OS 코스를 택했다면 하드웨어 장치 레지스터에 대한 사용자-레벨 억세스는 정확히 여러분이 한번도 배운 적이 없는 것이다. 왜냐면 OS의 주요 목적 중 하나는 장치 억세스를 제어하는 것이기 때문이다. 그러나 OS 호출은 적어도 수십 마이크로 초 오버헤드가 걸린다. 단지 3 마이크로 초 동안에 기본 네트웍 작업을 수행할 수 있는 TTL_PAPERS와 같은 커스텀 네트웍 하드웨어의 경우 그런 OS 호출 오버헤드는 참을 수 없는 것이다. 그런 오버헤드를 피하는 유일한 방법은 하드웨어 장치 레지스터들을 직접 억세스하는 사용자-레벨 코드 - 사용자-레벨 라이브러리 - 를 가지는 것이다. 그래서 사용자-레벨 라이브러리가 하드웨어를 직접 억세스할 수 있는 방법은 무엇인가, 하지만 장치 억세스 권한에 대한 OS 제어와 타협하지 않는 방법은 무엇인가와 같은 것이 질문이 될 것이다.

전형적인 시스템에서 사용자-레벨 라이브러리가 하드웨어 장치 레지스터를 직접 억세스하는 유일한 방법은 다음과 같다:

1. 사용자 프로그램 시작에서 장치 레지스터를 포함하는 메모리 주소 공간을 사용자 프로세스 가상 메모리 맵으로 맵핑하는 OS 호출을 사용한다. 어떤 시스템들에서는 `mmap()` 호출(섹션 2.6 (메모리 맵 호출) 에서 맨처음 언급됨)이 I/O 장치들의 물리적 메모리 페이지 주소들의 표현하는 특수 파일을 맵핑하는 데 사용될 수 있다. 또는 이런 기능을 수행하는 장치 구동기를 작성하는 일은 상대적으로 쉽다. 더 나아가 이 장치 구동기는 필요한 특정 장치 레지스터들을 담고 있는 페이지(들)을 맵핑하는 것만으로 억세스를 제어할 수 있다. 그래서 OS 억세스 제어를 유지할 수 있다.
2. 맵핑된 주소들에 단순하게 로딩하거나 저장함으로써 OS 호출 없이 장치 레지스터들을 억세스. 예를 들어서 `*((char *) 0x1234) = 5;`는 메모리 위치 1234(16진수)에다 바이트 값 5를 저장할 것이다.

다행스럽게도 인텔 386(그리고 호환 프로세스들)에 대한 리눅스가 좀 더 나은 솔루션을 제공하는 일이 벌어졌다:

1. 권한이 있는 프로세스로부터 `ioperm()` OS 호출을 사용함으로써 장치 레지스터에 대응하는 정확한 I/O 포트 주소들에 억세스하는 퍼미션을 얻는다. 또는 리눅스에 대한 패치 <http://garage.ecn.purdue.edu/~papers/giveioperm.html>을 사용하여 독립된 권한있는 사용자 프로세스(즉, "메타 OS")에 의해서 퍼미션이 관리될 수 있다.
2. 386 포트 I/O 명령어들을 사용해서 OS 호출 없이 장치 레지스터들을 억세스.

다수의 I/O 장치들이 단일 페이지 안에 그들의 레지스터를 갖는 것이 일반적이기 때문에 이 두번째 솔루션이 더 선호된다. 이런 경우 첫번째 기술은 의도된 것과 동일한 페이지에 위치하게 된 다른 장치 레지스터들을 억세스하지 못하도록 하는 보호를 제공하지 못할 것이다. 물론 386 포트 I/O 명령들이 C로 코딩될 수 없다는 것이 단점이다 - 대신 여러분은 약간의 어셈블리 코드를 사용할 필요가 생길 것이다. 바이트 값의 포트 입력을 위한 GCC-랩핑된(C 프로그램에서 사용 가능한) 인라인 어셈블리 코드 함수는 다음과 같다:

```
extern inline unsigned char
inb(unsigned short port)
{
    unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
                      :"=a" (_v)
                      :"d" (port), "0" (0));
    return _v;
}
```

비슷하게 바이트 포트 출력을 위한 GCC-랩핑된 코드는 다음과 같다:

```
extern inline void
outb(unsigned char value,
unsigned short port)
{
__asm__ __volatile__ ("outb %b0,%w1"
                      :/* no outputs */
                      :"a" (value), "d" (port));
}
```

3.4 PVM (병렬 가상 기계, Parallel Virtual Machine)

PVM(병렬 가상 기계)는 일반적으로 소켓 위에 구현된, 자유롭게 사용할 수 있고 이식될 수 있는 메시지 전달 라이브러리이다. 이것은 분명히 메시지 전달 클러스터 병렬 컴퓨팅을 위한 사실상의 표준으로 자리를 잡았다.

PVM은 단일-프로세서와 SMP 리눅스 기계들, 그리고 소켓-가능 네트웍(예, SLIP, PLIP, 이더넷, ATM)에 의해서 링크된 리눅스 기계들의 클러스터를 지원한다. 사실 PVM은 다양한 서로 다른 타입들의 프로세서들, 설정, 그리고 물리적인 네트웍들이 사용된 기계들 그룹 - **이기종 클러스터** -에서 병렬 클러스터로써 인터넷을 통해서 링크된 기계들을 처리하는 범위까지 작동한다. PVM은 또한 클러스터를 통해서 병렬 작업 제어를 위한 기능들을 제공한다. 이들 중 가장 좋은 것, PVM은 오랫동안 자유롭게 사용 가능하였고(현재는 `<http://www.epm.ornl.gov/pvm/pvm_home.html>`에 있음) 많은 프로그래밍 언어, 어플리케이션 라이브러리, 디버깅 툴 등과 같은 것에, 그것을 그들의 "이식 가능한 메시지-전달 타겟 라이브러리"로 사용하여, 이르게 되었다. 네트웍 뉴스 그룹 `comp.parallel.pvm`이 있다.

그러나 PVM 메시지 전달 호출들은 일반적으로 이미 높은 지체를 가지는 표준 소켓 작업들에 심각한 오버헤드를 추가한다. 더 나가아가 메시지 핸들링 호출들 자신은 특별히 "프렌들리"한 프로그래밍 모델을 이루지 않았다.

섹션 1.3 (예제 알고리즘)에서 맨처음 설명된 것과 동일한 파이(pi) 계산 예제를 사용해서 만든, C와 PVM 라이브러리 호출을 사용한 버전은 다음과 같다:

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

#define NPROC   4

main(int argc, char **argv)
{
  register double lsum, width;
  double sum;
  register int intervals, i;
  int mytid, iproc, msgtag = 4;
  int tids[NPROC];  /* array of task ids */

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* Join a group and, if I am the first instance,
     iproc=0, spawn more copies of myself
  */
  iproc = pvm_joingroup("pi");
```

```
  if (iproc == 0) {
    tids[0] = pvm_mytid();
    pvm_spawn("pvm_pi", &argv[1], 0, NULL, NPROC-1, &tids[1]);
  }
  /* make sure all processes are here */
  pvm_barrier("pi", NPROC);

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  lsum = 0.0;
  for (i = iproc; i<intervals; i+=NPROC) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }

  /* sum across the local results & scale by width */
  sum = lsum * width;
  pvm_reduce(PvmSum, &sum, 1, PVM_DOUBLE, msgtag, "pi", 0);

  /* have only the console PE print the result */
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }

  /* Check program finished, leave group, exit pvm */
  pvm_barrier("pi", NPROC);
  pvm_lvgroup("pi");
  pvm_exit();
  return(0);
}
```

3.5 MPI (메시지 전달 인터페이스, Message Passing Interface)

PVM은 사실상 표준 메시지-전달 라이브러리인 반면에 MPI(메시지 전달 인터페이스)는 상대적으로 새로운 공식 표준이다. MPI 표준에 대한 홈 페이지는 `<http://www.mcs.anl.gov:80/mpi/>`이며 뉴스그룹은 `comp.paralle.mpi`이다.

그러나 MPI를 논의하기 전에 저자는 지난 몇년 동안 일어난 PVM 대 MPI 종교 전쟁에 대해서 조금 얘기하고 싶은 충동을 느낀다. 다음은 차이점들에 대해서 상대적으로 편견없이 요약하려고 시도한 것이다:

실행 제어 환경(Execution control environment).

단순하게 얘기해서 MPI는 실행 제어 환경이 어떻게 구현되는가와 구현될 수 있는지 없는지를 지정하지 않은 반면 PVM은 실행 제어 환경 하나를 갖는다. 그래서 PVM 프로그램 실행을 시작하는 것과 같은 일들은 모든 곳에서 동일하게 이루어지는 반면 MPI의 경우 이것은 어떤 구현이 사용되는가에 따라서 다를 수 있다.

이기종 클러스터 지원(Support for heterogeneous clusters).

PVM은 워크스테이션 사이클-활용 세계에서 자라났고 그래서 직접 기계와 운영 체제의 이기종 혼합을 관리한다. 반면에 MPI는 타겟이 MPP(거대한 병렬 프로세서)이거나 거의 동일한 워크스테이션들의 전용 클러스터일 것이라고 가정한다.

부엌 싱크대 증후군(Kitchen sink syndrome).

PVM는 MPI 2.0이 하지 못하는 목적의 통일을 증명한다. 새로운 MPI 2.0 표준은 기본 메시지 전달 모델을 벗어나는 많은 기능들을 담고 있다 - RMA(리모트 메모리 억세스, Remote Memory Access)와 병렬 파일 I/O와 같은 것들. 이런 것들이 유용한가? 물론 그들은 그렇다... 그러나 MPI 2.0을 배우는 것은 완전히 새로운 프로그래밍 언어를 배우는 것과 거의 똑같다.

사용자 인터페이스 설계(User interface design).

MPI는 PVM을 따라 설계되었고 분명히 그것으로부터 배웠다. MPI는 더 단순하고 더 효과적인 버퍼 핸들링과 메시지로 사용자-정의 데이터 구조가 전달되도록 하는 고수준 추상화를 제공한다.

법의 효력(The force of law).

내 계산에 의하면 MPI를 사용하는 것보다 PVM을 사용하도록 설계된 것들이 아직 아주 많이 있다; 그러나 그것들을 MPI로 포팅하는 것은 쉽다. 그리고 MPI가 널리 지원되는 형식적인 표준에 의해서 지원된다는 사실은 MPI를 사용하는 것은 많은 기관들의 경우, 정책상의 문제이다는 것을 의미한다.

결론이 났는가? 글쎄 리눅스 시스템들로 만든 클러스터에서 실행할 수 있는, 자유롭게 사용할 수 있고 독립적으로 개발된 MPI 버전은 적어도 세가지 존재한다(그리고 나는 그것들 중 하나를 여기에서 설명한다):

- LAM(근거리 영역 멀티 컴퓨터, Local Area Multicomputer)는 MPI 1.1 표준을 완전히 구현한 것이다. 이것은 MPI 프로그램들이 개별 리눅스 시스템안이나 UDP/TCP 소켓 통신을 사용하는 리눅스 시스템 클러스터를 통해서 실행되는 것을 허용한다. 이 시스템은 단순한 실행 제어 기능들을 담고 있으며 다양한 프로그램 개발과 디버깅 도우미(aids)를 가지고 있다. 이것은 다음 `<http://www.osc.edu/lam.html>` 서 자유롭게 사용할 수 있다.
- MPICH(MPI 카멜레온, CHameleon)은 MPI 1.1 표준의 이식성이 높은 완전한 구현으로써 고안되었다. LAN과 같이 이것은 개별 리눅스 시스템이나 UDP/TCP 소켓 통신을 사용하는 리눅스 시스템들에 걸쳐서 MPI 프로그램들이 실행되도록 허용한다. 그러나 효율적이고 쉽게 타겟을 바꿀 수 있는 구현을 제공함으로써 명백하게 MPI를 증진한 것이 그 장점이다. 이 MPI 구현을 포팅하기 위해서 "채널 인터페이스(channel interface)"의 다섯가지 함수들을 구현하거나 좀 더 나은 성능을 위해서 완전한 MPICH ADI(추상 장치 인터페이스, Abstract Device Interface)를 구현한다. MPICH, 그리고 이것에 대한 것과 포팅에 관한 많은 정보들이 `<http://www.mcs.anl.gov/mpi/mpich/>`에 있다.
- AFMPI(집합 함수 MPI, Aggregate Function MPI)는 MPI 2.0 표준 구현의 부분 집합이다. 이것은 내가 작성한 것이다. AFAPI 위에 만들어졌고 낮은-지체시간 집합(collective) 통신 함수들과 RMA들을 보여주도록 고안되었고 그래서 MPI 데이터 타입들, 통신자들(communicators)등만을 제공한다. 이것은 개별 리눅스 시스템이나 AFAPI-가능 네트웍 하드웨어에 의해서 연결된 클러서터에 결쳐서 MPI를 사용한 C 프로그램들을 허락한다. 이것은 `<http://garage.ecn.purdue.edu/~papers/>`에서 자유롭게 얻을 수 있다.

이들 MPI 구현물들중 어떤 것을 쓰던 간에 대부분의 공용 통신 타입들을 수행하는 것은 아주 단순하다.

그러나 MPI 2.0은, 이들 중에 하나를 사용하는 프로그래머가 MPI와 같은 다른 코딩 스타일들을 인식하지 못할 정도로 충분히 서로 다른 여러가지 통신 패러다임들을 서로 연동시킨다. 그래서 단 하나의 예제 프로그램을 제공하는 것보다 MPI가 지원하는 기본적으로 서로 다른 통신 패러다임들 각각의 예제를 가지는 것이 유용하다. 아래에 나오는 모든 세가지 프로그램들은 이 HOWTO를 통해서 사용되는 Pi의 값을 구하는, 동일한 기본 알고리즘을 구현한다.

각 프로세서가 그것의 부분합을 총합을 구하고 그 결과를 출력하는 프로세서 0번에게 전달하기 위해서 기본 MPI 메시지-전달 호출들을 사용하는 첫번째 MPI 프로그램:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;
  MPI_Status status;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  if (iproc != 0) {
    MPI_Send(&lbuf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else {
    sum = lsum;
    for (i=1; i<nproc; ++i) {
      MPI_Recv(&lbuf, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      sum += lsum;
    }
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

두번째 MPI 버전은 집합적인(collective) 통신(이 특별한 어플리케이션의 경우 이것은 가장 적절하다):

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;
```

```
  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

세번째 MPI 버전은 각 프로세서가 자신의 로컬 `lsum`를 프로세서 0번의 `sum`로 더하기 위해서 MPI 2.0 RMA 메카니즘을 사용한다.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double width;
  double sum = 0, lsum;
  register int intervals, i;
  int nproc, iproc;
  MPI_Win sum_win;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  MPI_Win_create(&sum, sizeof(sum), sizeof(sum),
                 0, MPI_COMM_WORLD, &sum_win);
  MPI_Win_fence(0, sum_win);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
```

```
  MPI_Accumulate(&lsum, 1, MPI_DOUBLE, 0, 0,
                 1, MPI_DOUBLE, MPI_SUM, sum_win);
  MPI_Win_fence(0, sum_win);
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

MPI 2.0 RMA 메카니즘은, 서로 다른 메모리 위치들에 거주하는 다양한 프로세서들위의 대응하는 데이터 구조의 임의의 잠재적 문제점들을 아주 잘 극복한다는 점을 주목하는 것이 좋을 것이다. 이것은, 베이스 주소를 의미하는 "창(window)"를 참조하는 것과 경계 초월 억세스(out-of-bound access)에 대한 보호와 그리고 평등한 주소 크기 조절(even address scaling)에 의해서 가능하다. RMA 처리는 다음 `MPI_Win_fence` 이전까지 연기될 수 있다는 사실에 의해서 도움을 받는다. 간단히 말해서 RMA 메카니즘은 분산 공유 메모리와 메시지 전달 간의 이상한 조합(strange cross)이라고 말할 수도 있겠지만 잠재적으로 아주 효율적인 통신을 생성하는 아주 깨끗한 인터페이스이다.

3.6 AFAPI (집합 함수 API, Aggregate Function API)

PVM, MPI 등과 다르게 AFAPI(집합 함수 어플리케이션 프로그램 인터페이스; Aggregate Function Application Program Interface)는 현존하는 네트웍 하드웨어와 소프트웨어 위에 놓인 포팅가능한 추상 인터페이스를 구축하는 시도로써 일생을 시작하지 않았다. 그러기 보다는 AFAPI는 PAPERS (병렬 실행과 빠른 동기화를 위한 퍼듀 아답터; Purdue's Adapter for Parallel Execution and Rapid Synchronization; `<http://garage.ecn.purdue.edu/~papers/>` 참조)에 대한 하드웨어-종속적인 로우-레벨 지원 라이브러리로써 시작하였다.

PAPERS는 3.2 (네트웍 하드웨어)에서 약간 논의되었다; 이것은 지체시간이 약 수 마이크로 초동안인 공용 도메인 설계 커스텀 집합 함수 네트웍이다. 그러나 현존하는 슈퍼컴퓨터들보다 컴파일러 기술에서 더 나은 타겟이 될 수 있는 슈퍼컴퓨터를 만들려는 시도로써 이것이 개발되었다는 것이 PAPERS에 대해서 중요한 것이다. 이것은 질적으로 대부분의 리눅스 클러스터 노력들과, 상대적으로 별로 충분하지 않는 성긴 병렬 어플리케이션들에 대한 표준 네트웍을 사용하려고 노력하는 데 촛점을 맞춘 PVM/MPI들과 아주 다르다. 리눅스 피씨들이 PAPERS 시스템의 컴포넌트들로 사용된다는 사실은 단순하게 가능한한 가장 비용-효율적인 방식으로 프로토타입들을 구현해본 것이다는 것을 말할 뿐이다.

한 타스의 서로다른 프로토타입 구현물들에 대한 공용 로우-레벨 소프트웨어 인터페이스 필요가 AFAPI로 PAPERS 라이브러리가 표준화되도록 한 것이다. 그러나 AFAPI에 의해서 사용되는 모델은 타고날 적부터 더 단순하고, 전형적인 병렬처리 컴파일러에 의해서 컴파일된 코드나 SIMD 아키텍쳐들을 위해서 작성된 코드들의 더 정교한 상호작용에 좀 더 적당하다. 이 모델의 단순성은 PAPERS 하드웨어를 만들기 쉽게 할 뿐만 아니고 SMP들과 같은 다른 하드웨어 시스템들에 대한 놀랍도록 효율적인 AFAPI 포팅을 가능하게 한다.

AFAPI는 현재 TTL_PAPERS, CAPERS, 또는 WAPERS를 사용하여 커넥트된 리눅스 클러스터들에서 잘 작동한다. 이것은 또한 (OS 호출들이나 심지어 버스-락킹 명령어들 없이도, 섹션 2.2 (공유 메모리 프로그래밍에 대한 소개) 참조) SHMAPERS라고 불리는 시스템 V 공용 메모리 라이브러리(System V Shared Memory library)를 사용한 SMP 시스템들 위에서도 작동한다. 전통적인 네트웍(예, 이더넷) 위에서 UDP 브로드캐스트를 사용한 리눅스 클러스터 위에서 실행하는 버전이 개발 중에 있다. 모든 관련된 버전들이 `<http://garage.ecn.purdue.edu/~papers/>`에서 얻을 수 있을 것이다. AFAPI의 모든 버전들은 C나 C++로부터 호출되도록 설계되었다.

다음 예제는 섹션 1.3 (예제 알고리즘)에서 설명된 바 있는 Pi의 AFAPI버전이다.

```
#include <stdlib.h>
#include <stdio.h>
#include "afapi.h"

main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  if (p_init()) exit(1);

  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  sum = 0;
  for (i=IPROC; i<intervals; i+=NPROC) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }

  sum = p_reduceAdd64f(sum) * width;

  if (IPROC == CPROC) {
    printf("Estimation of pi is %f\n", sum);
  }

  p_exit();
  return(0);
}
```

3.7 다른 클러스터 지원 라이브러리들

PVM, MPI, 그리고 AFAPI에 덧붙여 다음 라이브러리들은 리눅스 클러스터를 사용해서 병렬 컴퓨팅하는데 유용할 기능들을 제공한다. 이런 시스템들은 이 문서에서 좀 더 가벼운 취급을 받았다. 왜냐면 PVM, MPI, 그리고 AFAPI와 다르게 나는 리눅스 클러스터에 이런 시스템들을 직접 사용할 기회가 전혀 또는 거의 없었기 때문이다.이런 것들이나 다른 라이브러리들 중 어떤 것이라도 특별히 유용하다고 생각되면 여러분이 찾은 것을 설명한 이메일을 `pplinux@ecn.purdue.edu`로 보내주기 바란다. 그러면 나는 그 라이브러리를 확장 섹션에 더하는 것을 고려하겠다.

3.7.1 Condor (프로세스 이주 지원, process migration support)

Condor는 워크스테이션들의 커다란 이기종 클러스터를 관리할 수 있는 분산 자원 관리 시스템이다. 이것의 설계 동기는 클러스터의 활용되지 않는 역량을 오래-실행되고 계산이 많은 일들에 대해서 사용하고자 하는 사람들의 요구에 의해서이다. Condor는 시각 및 실행 기계들이 공통 파일 시스템을 공유하지 않고/거나 패스워드 메카니즘을 공유하지 않을지라도, 실행 기계에 커다란 시작 기계의 환경을 유지한다. 단일 프로세스를 유지하는 Condor 작업들은 이벤트 완료를 확인하는 데 필요하기 때문에 자동으로 checkpoint되고 워크스테이션들 사이에서 이주된다.

Condor는 `<http://www.cs.wisc.edu/condor/>` 에서 찾을 수 있다. 리눅스 포팅된 버전이 존재한다; 더 자세한 내용은 `<http://www.cs.wisc.edu/condor/linux/linux.html>`을 찾아 볼 수 있다. 자세한 것은

condoradmin@cs.wisc.edu을 만나보라.

3.7.2 DFN-RPC (German Research Network - Remote Procedure Call)

DFN-RPC(독일 연구 네트웍 RPC) 툴은 워크스테이션과 계산 서버나 클러스터 간에 과학-기술적인 어플리케이션 프로그램들을 분산하고 병렬화(parallelize)하기 위해서 개발되었다. 이 인터페이스는 포트란으로 작성된 어플리케이션에 최적화되어 잇지만 DFN-RPC는 또한 C 환경에서도 사용될 수 있다. 이것은 리눅스로 포팅되었다. 자세한 정보는 <ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html>에 있다.

3.7.3 DQS (분산 큐잉 시스템, Distributed Queueing System)

정확하게 말해서 라이브러리는 아니지만 DQS 3.0 (분산 큐잉 시스템)은 리눅스에서 개발되고 테스트된 작업 큐잉 시스템이다. 이것은 이기종 클러스터를 하나의 엔터티로써 사용하고 관리할 수 있도록 설계되었다. 이것은 <http://www.scri.fsu.edu/~pasko/dqs.html>에서 얻을 수 있다.

CODINE 4.1.1(분산 네트웍 환경에서의 계산, COmputing in DIstributed Network Environments)라고 불리는 상업용 버전도 또한 존재한다. 이것의 정보는 <http://www.genias.de/genias_welcome.html>에서 찾을 수 있다.

3.8 일반 클러스터 참고자료

클러스터들은 아주 많은 방식으로 구축되고 사용될 수 있기 때문에 흥미로운 공헌을 한 그룹들이 꽤 있다. 다음은 일반적인 관심거리가 될 수 있는 다양한 클러스터-관련 프로젝트들에 대한 레퍼런스이다. 이것은 리눅스에 한정된 것과 그렇지 않은 일반적인 클러스터 레퍼런스들을 담고 있다. 이 리스트는 알파벳 순서로 나와 있다.

3.8.1 Beowulf

Beowulf 프로젝트는, <http://cesdis1.gsfc.nasa.gov/beowulf/>, 상품 피씨-클래스 하드웨어, 고-대역폭 클러스터-인터널 네트웍, 그리고 리눅스 운영 체제를 기반으로 한 규격품 워크스테이션 클러스터를 사용하기 위한 소프트웨어 제작에 집중하고 있다.

Thomas Sterling가 Beowulf 뒤의 추진력이었으며 계속해서 일반적인 과학 컴퓨팅을 위한 리눅스 클러스터링의 웅변적이고 솔직한 제안자이었다. 사실, 많은 그룹들의 지금 그들의 클러스터를 "Beowulf class" 시스템들이라고 부른다 - 심지어 그 클러스터가 실제 공식적인 Beowulf 설계에 전혀 비슷하지 않더라도 말이다.

Beowulf 프로젝트를 지원하는 일을 하는 Don Becker는 리눅스에 의해서 일반적으로 사용되는 많은 네트웍 드라이버들을 만들었다. 이들 중 많은 것들이 BSD에서 사용되도록 적용되어 왔다. Don은 또한 비싼 스위치 허브없이 더 높은 대역폭을 획득하도록 다수의 병렬 커넥션들을 통해서 로드-분배를 허용하는 많은 리눅스 네트웍 드라이버들에 대해서도 책임을 지고 있다. 이런 로드-분배(load-sharing) 타입은 Beowulf 클러스터가 원조인 다른 것과 구별된는 기능이었다.

3.8.2 Linux/AP+

Linux/AP+ 프로젝트, <http://cap.anu.edu.au/cap/projects/linux/>는 정확하게 리눅스 클러스터링에 대한 것이 아니고 리눅스를 Fujitsu AP1000+으로 포팅하는 것과 적절한 병렬 처리 증진을 더하는 것에 촛점을 맞춘 것이다. AP1000+는 환형 토폴로지, 25 MB/s 대역폭, 그리고 10 마이크로 초 지체 시간 ... 을 가진 커스텀 네트웍을 사용하는 상용 SPARC-기반 병렬 기계이다. 단순하게 말해서 이것은 SPARC 리눅스 클러스토와 아주 유사하다.

3.8.3 Locust

로커스트(Locust) 프로젝트, `<http://www.ecsl.cs.sunysb.edu/~manish/locust/>`는 메시지-지체시간을 숨기기 위해서 그리고 실시간으로 네트웍 트래픽을 줄이기 위해서 컴파일-시간 정보를 사용하는 분산 가상 공유 메모리 시스템을 구축하고 있다. Pupa는 로커스트의 기반 통신 서브시스템이고 FreeBSD의 486 피씨들을 연결하기 위해서 이더넷을 사용하여 구현되었다. 리눅스인가?

3.8.4 Midway DSM (Distributed Shared Memory)

Midway, `<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/midway/WWW/HomePage.html>`는 TreadMarks와 다르지 않는, 소프트웨어-기반 DSM(분산 공유 메모리) 시스템이다. 이것은 상대적으로 느린 페이지-폴트 메카니즘들보다는 컴파일-시간 도움(aids)을 사용하고 공짜라는 것이 희소식이다. 나쁜 소식은 이것이 리눅스 클러스터들 위에서 작동하지 않는다는 것이다.

3.8.5 Mosix

MOSIZ는 BSDI BSD/OS를 변조해서 피씨들을 네트웍으로 묶은 그룹 위에서 동적 로드 밸런싱과 선점(preemptive) 프로세스 이주를 제공하도록 한 것이다. 이것은 병렬 처리에 대해서뿐만이 아니고 조절가능한(scalable) SMP와 아주 유사한 클러스터를 일반적으로 사용하는 데에도 좋은 것이다. 리눅스 버전이 있을까? 좀 더 자세한 정보는 `<http://www.cs.huji.ac.il/mosix/>`를 참조하자.

3.8.6 NOW (Network Of Workstations)

버클리 NOW (네트웍으로 묶은 워크스테이션들, Network Of Workstations) 프로젝트, `<http://now.cs.berkeley.edu/>`는 네트웍으로 묶은 워크스테이션들을 사용해서 병렬 컴퓨팅을 하도록 많은 압력을 행사해왔다. 현재 많은 작업들이, 모두 "다음 몇년 안에 실질적인 100개의 프로세서 시스템을 데모하는 것"을 향해서, 이루어지고 있다. 아뿔사, 이들은 리눅스를 사용하지 않는다.

3.8.7 리눅스를 사용하는 병렬 처리(Parallel Processing Using Linux)

리눅스를 사용한 병렬 처리 WWW 사이트, `<http://yara.ecn.purdue.edu/~pplinux/>`, 는 이 HOWTO의 홈 페이지이며 하루짜리 튜터리얼을 노린 온라인 슬라이들을 포함한 많은 관련된 문서들이 있는 곳이다. PAPERS 프로젝트에 대한 작업을 제외하고도 퍼듀 대학교 전자 및 컴퓨터 공학 학교는 일반적으로 병렬 처리의 리더로 군림해왔다; 이 사이트는 다른 사람들이 리눅스 피씨들을 병렬 처리에 적용하는 것을 돕기 위해서 설립되었다.

퍼듀의 첫번째 리눅스 피씨 클러스터가 1994년 2월에 조립된 이래로 비디오 벽(video wall)을 포함한, 수많은 리눅스 피씨 클러스터들이 퍼듀에서 조립되어왔다. 비록 이런 클러스터들이 386, 486, 그리고 Pentium 시스템들(Pentium Pro 시스템은 없다)을 사용했지만 요근래 인텔이 Pentium II 시스템들로 된 다수의 커다란 클러스터들을 만들 수 있도록 퍼듀 대학교에 기부를 했다. 이런 클러스터들이 모두 PAPERS 네트웍을 가지고 가질 것이지만 대부분의 것들이 또한 전통적인 네트웍을 가진다.

3.8.8 펜티엄 프로 클러스터 워크샵(Pentium Pro Cluster Workshop)

아이오와(Iowa) 주 데모인(Des Moines)에서 1997년 4월 10-11에 AMES 랩은 펜티엄 프로 클러스터 워크샵을 열었다. 이 워크샵의 WWW 사이트, `<http://www.scl.ameslab.gov/workshops/PPCworkshop.html>`, 는 모든 참가자들로부터 수집된 풍부한 피씨 클러스터 정보들을 갖고 있다.

3.8.9 TreadMarks DSM (Distributed Shared Memory)

DSM (분산 공유 메모리, Distributed Shared Memory)는 이것에 의해서 메시지-전달 시스템이 SMP처럼 행동하는 것같이 보일 수 있는, 기술이다. 몇가지 그런 시스템들이 존재하고 이들 중 대부분은 메시지 전송을 트리거(촉발)하기 위해서 OS 페이지-오류 메카니즘을 사용한다. TreadMarks, `<http://www.cs.rice.edu/~willy/TreadMarks/overview.html>`는 이런 시스템들 중 좀 더 효율적인 것이며 리눅스 클러스터 위에서 실행된다. 슬픈 소식은 "TreadMarks가 대학교들이나 비영리 기관들에 저비용으로 배포되고 있다는 것이다". 이 소프트웨어에 대한 좀 더 많은 정보를 원한다면 `treadmarks@ece.rice.edu`과 접촉해보기 바란다.

3.8.10 U-Net (User-level NETwork interface architecture)

코넬 대학교의 U-Net (User-level NETwork interface architecture) 프로젝트, `<http://www2.cs.cornell.edu/U-Net/Default.html>`는 네트웍 인터페이스를 가상화해서(virtualize) 어플리케이션들이 메시지들을 운영체제의 간섭없이 전송하거나 수신할 수 있도록 하는, 상품 네트웍 하드웨어를 사용한 낮은 지체 시간과 높은 대역폭을 제공하는 시도를 하고 있다. U-Net은 Fast 이더넷에 기반한 DECchip DC21140 를 사용하거나 Fore Systems PCA-200(PCA-200E가 아님) ATM 카를 사용하는 리눅스 피씨들 위에서 작동한다.

3.8.11 WWT (Wisconsin Wind Tunnel)

위스콘신에는 상당히 많은 클러스터-관련 작업들이 이루어지고 있다. WWT (Wisconsin Wind Tunnel) 프로젝트, `<http://www.cs.wisc.edu/~wwt/>`는 컴파일러들과 기반 패러렐 하드웨어 사이의 "표준" 인터페이스를 개발하는 쪽의 모든 종류의 일들을 수행하고 있다. Wisconsin COW (Cluster Of Workstations), Cooperative Shared Memory and Tempest, Paradyn Parallel Performance Tools 등이 있다. 불행하게도 리눅스에 관한 일들은 별로 없다.

제 4 절 하나의 레지스터위에서의 SIMD(예: MMX 사용)

단일 레지스터 내의 SIMD(SWAR)는 새로운 아이디어가 아니다. k-비트 레지스터, 데이터 패스, 그리고 함수 유닛 을 가지는 기계가 있을 때 일반 레지스터 연산들이 n개의 k/n-비트, 정수 필드 값들 위에서 SIMD 병렬 연산들로 수행될 수 있다는 것이 오래전부터 알려져 왔다. 그러나 SWAR 기술들에 의해서 제공되는 2배에서 8배까지의 속도 증가가 메인스트림 컴퓨팅에 대한 관심사가 된 것은 요즘들어 멀티미디어에 대한 강렬한 추세때문이다. 마이크로프로세서들의 1997 버전 대부분은 SWAR에 대한 하드웨어 지원을 담고 있다:

- `AMDK6MMX(MultiMediaeXtensions)`
- `CyrixM2MMX(MultiMediaeXtensions)`
- `DigitalAlphaMAX(MultimediAeXtensions)`
- `Hewlett-PackardPA-RISCMAX(MultimediaAccelerationeXtensions)`
- `IntelPentiumII\&PentiumwithMMX(MultiMediaeXtensions)`
- `MicrounityMediaprocessorSIGD(SingleInstructiononGroupsofData)`
- `MIPSDigitalMediaeXtension(MDMX,pronouncedMadMax)`
- `SunSPARCV9VIS(VisualInstructionSet)`

새로운 마이크로프로세서들에 의해서 제공되는 하드웨어 지원에는 몇가지 결함들이 있고 어떤 필드 크기들에 대해서 어떤 연산들만을 지원하는 것과 같은 단점이 있다. 그러나 많은 SWAR 연산들이 효율적이어야 하는 하드웨어 지원이 필요없다는 것을 기억하는 것이 중요하다. 예를 들어서 비트단위 연산들은 하나의 레지스터를 논리적으로 분할하는 것에 의해서 영향을 받지 않는다.

4.1 SWAR: 어디에 좋은 것이가(What Is It Good For)?

비록 모든 현재 프로세서들이 적어도 어떤 SWAR 병렬 기능(parallelism)과 함께 수행하는 것이 가능하다고 하더라도 가장 좋은 SWAR-개선 명령 셋들도 아주 일반적인-목적의 병렬 기능(parallelism)을 지원하지 않는다는 것이 슬픈 사실이다. 사실, 많은 사람들이 Pentium과 "Pentium with MMX technology" 사이의 수행 능력 차이가 우연히도 MMX와 동시에 나타난 더 큰 L1 캐쉬와 같은 것들 때문이라고 생각한다. 그렇다면 SWAR(즉 MMX)는 현실적으로 어디에 좋은 것인가?

- 정수들만 있는 것이, 더 적은 것이 더 좋다. 32-비트 값들은 64-비트 MMX 레지스터에 맞아 떨어지지만 8개의 1-바이트 문자들이나 1-비트값들의 체스판 크기(체스판은 가로 세로 8개씩의 칸이 있다. 그래서 64개; 역자주) 개수 전체이더라도 맞아 떨어진다.

  Note: 이 글을 쓰고 있는 순간 비록 많은 내용이 없지만 *MMX의 부동-소숫점 버전이 있을* 것이다. Cyrix는 **MMFP**에 대한 몇가지 설명을 담은 슬라이드들을 `<ftp://ftp.cyrix.com/developr/mpf97rm.pdf>`에 포스팅해놓았다. 얼른 봐서 MMFP는 두 32-비트 부동-소숫점 숫자들이 64-비트 MMX 레지스터 하나에 팩킹되는 것을 지원할 것이다; 이것을 두개의 MMFP 파이프라인들로 묶으면 클럭 하나당 네개의 단일-정밀도 FLOP를 생산할 것이다.

- SIMD, 또는 벡터-스타일 병렬 처리(parallelism). 동일한 연산이 모든 필드들에 동시에 적용된다. 선택된 필드들에 효과를 없애는 방법들(즉, SIMD enable 마스킹)이 존재하지만 그들은 복잡하고 성능을 저하시킨다.

- 지역화(localized)되고 정규적인(오히려 팩킹된), 메모리 참조 패턴들. 일반적인 SWAR, 그리고 특별히 MMX, 는 랜덤-순서 접근에 취약이다; 벡터 `x[y]`(여기서 `y`는 인덱스 배열이다)를 모으는 것은 엄청 비싸다.

이들은 심각한 제약점들이지만 이런 타입의 병렬 처리는 많은 병렬 알고리즘들에서 나타나는 것이다 - 멀티미디어 어플리케이션들뿐만이 아니고. 알고리즘의 적절한 타입에 대해서 SWAR는 SMP나 클러스터 병렬 처리보다 훨씬 효율적이다... 그리고 이것은 그것을 사용하는 데 어떤 추가 비용도 들지 않는다.

4.2 SWAR 프로그래밍에 대한 소개(Introduction To SWAR Programming)

SWAR의 기본 개념(컨셉), 단일 레지스터 안에서의 SIMD, 는 워드-길이 레지스터들 위에서의 연산들이 n개의 k/n-비트 필드 수치들에 대한 SIMD 병렬 연산을 수행함으로써 계산 속도를 높이는 데 사용될 수 있다는 것이다. 그러나 SWAR 기술을 사용하는 것은 다소 어색할 수 있으며 또한 어떤 SWAR 연산들은 실제 대응되는 일련의 순차적인 연산들보다, 그들이 필드 분할을 수행하는 추가의 명령어들을 요구하기 때문에, 더 고 비용이다.

이 관점을 예시하기 위해서 상당히 단순화된, 각 32-비트 레지스터 안에서 4개의 8-비트 필드들을 관리하는, SWAR 메카니즘을 생각해보도록 하자. 두 레지시터들안의 수치들은 다음과 같이 표현될 수 있겠다:

```
         PE3      PE2      PE1      PE0
      +-------+-------+-------+-------+
Reg0  | D 7:0 | C 7:0 | B 7:0 | A 7:0 |
      +-------+-------+-------+-------+
Reg1  | H 7:0 | G 7:0 | F 7:0 | E 7:0 |
      +-------+-------+-------+-------+
```

이것은 단순하게 각 레지스터가 기본적으로 4개의 독립 8-비트 정수 수치들의 벡터로 볼 수 있다는 것을 의미한다. 또는 항목 0 (PE0)를 처리하는 Reg0와 Reg1에 있는 수치들로써 A와 E를, 그리고 PE1의 레지스터들에 있는 수치들로써 B과 F를, 그리고 계속 이런 식으로 생각할 수 있다.

이 문서의 나머지 부분은 이런 정수 벡터들에 대한 SIMD 병렬 연산을 위한 기본 클래스들과 이들의 함수들이 어떻게 구현될 수 있는 가에 대해서 대략적으로 리뷰할 것이다.

4.2.1 다형성 연산(Polymorphic Operations)

어떤 SWAR 연산들은, 이 연산이 실제 이런 8-비트 필드들에 병렬로 서로 독립적으로 계산되도록 의도되었다는 사실에 신경쓰지 않고서 일반 32-비트 정수 연산들을 사용해서 쉽게 수행될 수 있다. 우리는 임의의 이런 SWAR 연산들을 *다형성(polymorphic)*이라고 부른다. 왜냐면 이 기능이 필드 타입들(크기들)에 의해서 영향을 받지 않기 때문이다.

임의의 필드가 영이 아닌가를 테스트하는 것은 다형성이다. 그리고 모든 비트단위 논리 연산들도 그렇다. 예를 들어서 일반 비트단위 and 논리 연산(C의 & 연산자)은 비트단위로 수행하고 그 필드 크기가 얼마나 되는지에 신경쓰지 않는다. 위 레지스터들의 단순한 비트단위 and 연산 결과는 다음과 같은 결과를 만들어 낸다:

```
         PE3       PE2       PE1       PE0
      +---------+---------+---------+---------+
Reg2  | D&H 7:0 | C&G 7:0 | B&F 7:0 | A&E 7:0 |
      +---------+---------+---------+---------+
```

비트단위 and 연산은 항상 연산대상 비트 k 수치들에 의해서만 영향을 받는 결과 비트 k의 수치를 가지기 때문에 어떤 필드 크기라도 동일한 단일 명령에 의해서 지원받는다.

4.2.2 분할된 연산(Partitioned Operations)

불행하게도 많은 중요한 SWAR 연산들이 다형성이 아니다. 더하기, 빼기, 곱하기, 나누기 등과 같은 사칙연산들은 모두 필드들간의 자리올림(carry)/빌려옴(borrow) 상호작용을 할 수밖에 없다. 우리는 이런 SWAR 연산들을 *분할된(partitioned)* 것이라고 부른다. 왜냐면 각 연산이 반드시 연산대상들을 효율적으로 분할해야 하고 필드들 간의 상호작용을 막아야 하기 때문이다. 그러나 이런 효과를 얻는 데 사용될 수 있는 세가지 서로 다른 방법들이 존재한다.

분할된 명령어(Partitioned Instructions) 분할된 연산들을 구현하는 가장 명백한 접근법은 필드들 간의 carry/borrow 논리를 자르는 "분할된 병렬 명령어"에 대한 하드웨어 지원을 제공하는 것이다. 이런 접근은 최고의 성능을 내지만 프로세서의 명령 집합을 변경해야 하고 일반적으로 필드 크기에 많은 제한들이 있다(예, 8-비트 필드들이 지원될 수 있지만 12-비트 필드들은 그렇지 못한 경우).

AMD/Cyrix/Intel MMX, Digital MAX, HP MAX, 그리고 Sun VIS는 모두 분할 명령들의 제한된 버전들을 구현한 것들이다. 불행하게도 이런 서로 다른 명령 셋 확장들은 중요한 다른 제약들을 가지기 때문에 그들 간에 알고리즘들이 서로 포팅될 수 없게 만든다. 예를 들어서 다음과 같은 분할된 연산의 샘플링을 생각해 보자:

```
  Instruction           AMD/Cyrix/Intel MMX   DEC MAX   HP MAX   Sun VIS
+---------------------+---------------------+---------+--------+---------+
| Absolute Difference |                     |       8 |        |       8 |
+---------------------+---------------------+---------+--------+---------+
| Merge Maximum       |                     |   8, 16 |        |         |
```

Compare	8, 16, 32			16, 32
Multiply	16			8x16
Add	8, 16, 32		16	16, 32

이 테이블에서 숫자들은 각 연산이 지원되는 필드 크기들을 비트 단위로 나타낸 것이다. 비록 이 테이블이 좀 더 훌륭한 것들을 포함한 많은 명령들을 생략한 것이기는 하지만 많은 차이가 있다는 것은 분명한 사실이다. 이의 직접적인 결과는 고-수준 언어들(High-Level Languages; HLLs)가 실제로 프로그래밍 모델로써 아주 적합한 것은 아니다라는 것과 포팅이 일반적으로 아주 나쁘다는 것이다.

교정 코드를 가지는 분할되지 않은 연산(Unpartitioned Operations With Correction Code) 분할 명령어들을 사용해서 분할 연산들을 구현하는 것은 분명히 효율적일 수 있지만 필요한 분할 연산이 하드웨어에 의해서 지원되지 않으면 어떻게 할 것인가? 해답은 필드간 carry/borrow을 가진 연산들을 일반 명령어들을 사용해서 수행하고 원하지 않는 필드 상호작용을 교정하는 것이다.

이것은 순전히 소프트웨어로 접근하는 것이고 교정작업은 오버헤드를 일으키지만 완전히 일반적인 필드 분할로 잘 작동한다. 이런 접근법은 분할 명령에 대한 하드웨어 지원의 갭들을 채우는 데 사용될 수 있거나 아니면 하드웨어 지원을 전혀 하지 않는 타겟 기계들에 대해서 완전한 기능을 제공하는 데 사용될 수 있다는 점에서 또한 완전히 일반적이다. 사실 C와 같은 언어로 코드 시퀀스들을 표현함으로써 이런 접근법은 SWAR 프로그램들이 완전히 포팅 가능한 것으로 만든다.

그렇다면 다음과 같은 질문이 바로 생긴다: 비분할 연산들을 교정 코드로 SWAR 분할 연산들을 시뮬레이션하는 것이 정확히 얼마나 비효율적인가? 글쎄 이것은 확실히 $64k 문제이다... 하지만 많은 연산들이 예상하는 것만큼 어려운 것은 아니다.

일반적인 32-비트 연산들을 사용해서 네개의 성분을 가지는 8-비트 정수 벡터들 두개를 더하는 것, `x+y`을 생각해보자.

일반적인 32-비트 덧셈은 실제로 정확한 결과를 만들지만 8-비트 필드들 중 하나라도 다음 필드로 캐리(자리 올림)를 만든다면 정확한 결과를 만들어내지 못한다. 그래서 우리의 목적은 단순하게 그런 캐리가 일어나지 않도록 보장하는 것이다. 두개의 k-비트 필드들을 더하는 것은 많아야 k+1 비트 결과를 만들어 내기 때문에 우리는 각 필드의 msb(most significant bit)를 단순히 "마스킹 제거(masking out)"함으로써 어떤 캐리도 발생하지 않도록 보장할 수 있다. 이것은 `0x7f7f7f7f`로 각 피연산자를 비트단위 and(bitwise anding)하고 나서 일반 32-비트 더하기를 수행함으로써 이루어진다.

```
t = ((x & 0x7f7f7f7f) + (y & 0x7f7f7f7f));
```

이 결과는 정확하다... 각 필드의 msb를 제외하고 말이다. 각 필드에 대해서 교정값을 계산해보자. 이것은, `x`와 `y`의 msb들을 `t`에 대해서 계산된 7-비트 캐리 결과에 두 개의 1-비트 분할된 덧셈을 하는 문제에 지나지 않는다. 다행스럽게도 1-비트 분할 덧셈은 일반 exclusive or 연산으로 구현되어 있다. 그래서 그 결과는 다음과 같다:

```
(t ^ ((x ^ y) & 0x80808080))
```

좋다, 글쎄, 이것은 그렇게 단순한 것이 아닐 수 있다. 결국 4개의 덧셈을 위해서 6번 연산을 수행한다. 그러나 연산이 횟수는 필드가 몇개인가에 따라 다르지 않다는 것을 주목하자. 그래서 좀 더 많은 필드들이 있으면 우리는 속도 향상을 얻을 수 있다. 사실 필드들이 단일 연산(정수 벡터)으로 로드되고 저장되었기 때문에, 우리는 어떤 식으로든 단순하게 속도 향상할 수 있으며, 레지스터 가용성은 개선될 수 있고, 동적 코드 스케줄링 종속성이 더 적다(부분 워드 참조를 피할 수 있기 때문에).

필드 수치 제어(Controlling Field Values) 부분 연산 구현에 대한 다른 두가지 접근법 둘 다 레지스터들에 대한 공간 활용을 최대화하려고 하는 반면에, 대신 필드 값들을 제어해서 내부-필드 캐리/빌림 이벤트들이 절대 일어나지 않도록 하는 것이 좀 더 계산 측면에선 효율적이다. 예를 들어서 우리가 더해진 모든 필드 값들이 어떤 필드 오버플로우도 일어나지 않는다는 것을 안다면 부분 더하기 연산은 일반적인 더하기 명령을 사용해서 구현될 수 있다; 사실 이런 제한이 주어지면 일반적인 더하기 연산이 다형성(역자주: 필드 크기에 독립이다)인 것처럼 보이고 교정 코드 없이 어떤 필드 크기들에도 사용 가능하다. 그래서 어떻게 필드 값들이 캐리/빌림 이벤트를 발생시키지 않도록 보장할 수 있는가가 관건이 된다.

이런 특성을 보장하는 한 가지 방법은 필드 값들의 범위를 제한할 수 있는 부분화된 명령들을 구현하는 것이다. Digital MAX 벡터 minimum과 maximum 명령들은 내부-필드 캐리/빌림을 피하기 위해서 필드 값들을 클립핑(역자주: 자름)하는 하드웨어적인 지원이다.

그러나 우리가 필드 값들의 범위를 효과적으로 제한할 수 없는 부분화된 명령들을 가지지 못한다고 가정하자... 값싸게 캐리/빌림 이벤트들이 인접 필드들 사이에 간섭하지 않는다고 보장하도록 할 수 있는 충분한 조건이 있는가? 이의 해답은 사칙연산 특성의 분석에 있다. 두 k-비트 숫자들을 더하는 것은 많아야 k+1 비트로 된 숫자를 생성한다; 그래서 k+1 비트는 일반 명령들을 사용함에도 불구하고 그런 연산을 안전하게 담을 수 있다.

그래서 우리의 이전 예제안에서 8-비트 필드들이 이제는 1-비트의 "캐리/빌림 완충기(spacers)"를 가지는 7-비트 필드들이라고 가정하자:

```
             PE3          PE2          PE1          PE0
      +----+-------+----+-------+----+-------+----+-------+
Reg0  | D' | D 6:0 | C' | C 6:0 | B' | B 6:0 | A' | A 6:0 |
      +----+-------+----+-------+----+-------+----+-------+
```

7-비트 덧셈의 벡터는 다음과 같이 수행된다. 어떤 부분 연산을 시작하기 이전에 모든 캐리 완충 비트들(`A'`, `B'`, `C'`, 그리고 `D'`)가 0이라는 값을 갖는다고 가정하자. 단순하게 일반 덧셈 연산을 수행함으로써 모든 필드들은 정확한 7-비트 값들을 얻는다; 그러나 어떤 완충 비트 값들은 이제 1이 될 수 있다. 우리는 이것을 전통적인 연산인 완충 비트들에 대한 마스크-제거를 한번 더 수행함으로써 교정할 수 있다. 우리의 7-비트 정수 벡터 덧셈, `x+y`은 그래서 다음과 같다:

```
((x + y) & 0x7f7f7f7f)
```

이것은 네개의 덧셈을 두 명령어로 줄인 것이다. 그래서 이것은 좋은 속도 향상을 분명히 가져올 것이다.

주의 깊은 독자(sharp reader)는 완충 비트들을 0으로 설정하는 것은 빼기 연산에서 작동하지 않는다는 것을 눈치챘을 것이다. 그러나 그 교정 방법이 아주 단순하다. `x-y`를 계산하기 위해서 우리는 `x`에 있는 완충 비트들은 모두 1이고 `y`에 있는 완충 비트들은 모두 0이라는 초기 조건을 확실하게 한다. 가장 나쁜 경우에 우리는 다음과 같은 것을 얻을 것이다:

```
(((x | 0x80808080) - y) & 0x7f7f7f7f)
```

그러나 추가의 비트별 or 연산은 종종, `x`의 값을 생성하는 연산이 `& 0x7f7f7f7f` 대신에 `| 0x80808080`을 마지막 스텝으로써 사용한다는 것을 확실하게 함으로써, 최적화될 수 있다.

어떤 방법이 SWAR 부분화된 연산들에 대해서 사용되어야 할 것인가? 그 답은 단순하게도 "가장 빠른 속도(향상)을 내는 것이면 무엇이든 된다"는 것이다. 흥미롭게도 사용하기 위한 이상적인 방법은 동일한 기계위에서 동작하는 동일한 프로그램 내에서(도) 서로 다른 필드 크기들에 대해서 서로 다를 수 있다.

4.2.3 통신과 타입 변환 연산(Communication & Type Conversion Operations)

비록, 이미지 픽셀들에 대한 많은 연산들을 포함해서, 어떤 병렬 계산은 한 벡터의 i번째 값은 피연산자 벡터들의 i번째 위치에 나타나는 값들만의 함수이라는 속성을 갖고 있지만, 이것은 일반적으로 그런 경우가 아니다. 예를 들어서 부드럽게 하기(smoothing)와 같은 픽셀 연산들조차 인접 픽셀들을 피연산자들로 요구하고 FFT들과 같은 변환들도 좀 더 복잡한(덜 지역화된) 통신 패턴들을 요구한다.

SWAR를 위한, 부분화되지 않은 쉬프트 연산들을 사용한, 1-차원 가장 근접한 이웃 통신을 효율적으로 구현하는 것은 어려운 일이 아니다. 예를 들어서, PEi로부터 PE(i+1)로 값을 이동하기 위해서 단순한 쉬프트 연산으로도 충분하다. 필드들이 8-비트의 길이를 가진다면 다음과 같이 사용할 것이다:

```
(x << 8)
```

그러나 이것은 항상 그렇게 단순하지 않다. 예를 들어서 PEi로부터 PE(i-1)로 값을 이동하려면, 단순한 쉬프트 연산으로도 충분하다. 그러나 C 언어는 오른쪽 쉬프트가 부호 비트를 보존하는지 않하는지를 지정하지 않고 어떤 기계들은 부호 붙은 오른쪽 쉬프트만을 지원한다. 그래서 일반적인 경우 우리는 반드시 명시적으로, 잠재적인 복사된(replicated) 부호 비트들을 0으로 만들어야 한다:

```
((x >> 8) & 0x00ffffff)
```

"wrap-around 커넥션들"을 더하는 것도 또한 부분화되지 않은 쉬프트를 사용해서 상당히 효율적이다. 예를 들어서 PEi로부터 값을 PE(i+1)로 wraparound를 이용해서 옮기려면:

```
((x << 8) | ((x >> 24) & 0x000000ff))
```

실질적인 문제는 좀 더 일반적인 통신 패턴이 반드시 구현되어야 한느 경우에 발생한다. 단지 HP MAX 명령어 집합만이 단일 명령으로 필드들의 임의 재배치를 지원한다. 이것은 `Permute`라고 불린다. 이 `Permute` 명령은 실제로 이름이 잘못 지어졌다; 이것은 필드들의 임의의 permutation[1]만 수행하는 것이 아니라 반복(repetition)도 허용한다. 간단히 말해서 이것은 임의의 `x[y]` 연산을 수행한다.

불행하게도 `x[y]`는 그런 명령없이 구현하기가 아주 어렵다. 코드 시퀀스는 일반적으로 길면서도 비효율적이다; 사실 이것은 순차적인 코드이다. 이것은 아주 실망스러운 것이다. MasPar MP1/MP2와 Thinking Machines CM1/CM2/CM200 SIMD 슈퍼컴퓨터에서의 `x[y]`의 상대적으로 높은 연산 속도는 이런 기계들의 성능이 좋았던 주요 이유들 중의 하나이었다. 그러나 `x[y]`는 항상 가장 근접한 이웃 통신보다도, 심지어 그런 슈퍼컴퓨터들에서조차, 더 느리기 때문에 많은 알고리즘들이 `x[y]` 연산들에 대한 수요를 최소화하기 위해서 고안되어 왔었다. 간단하게 말해서 하드웨어 지원없이 이것은 `x[y]`가 합법적이지 않은 것처럼 또는 적어도 싼 것이 아닌것처럼 SWAR 알고리즘들을 개발하는 것이 가장 좋을 것이다.

4.2.4 순환 연산(Recurrence Operations) (축소, 스캔 등)

순환이란 계산되는 값들간의 외면상 순차적인 관계가 있는 계산을 말한다. 그러나 이런 순환이 결합적인 연산들을 포함한다면 세개의 구조화된 병렬 알고리즘을 사용하여 그 계산을 재코딩하는 것이 가능할 수 있다.

병렬화가 가능한 순환(recurrence)의 대부분의 일반적인 타입은 아마도 결합 축소(associative reduction)으로 알려진 클래스일 것이다. 예를 들어서 어떤 벡터 값들의 덧셈을 계산하기 위해서 다음과 같은 완전히 순차적인 C 코드를 작성하는 것이 일반적이다:

```
t = 0;
for (i=0; i<MAX; ++i) t += x[i];
```

[1]역자주: 순열이라고 번역하지만 수학에서는 일정 개수의 객체들의 자리 이동을 말한다

그러나, 이런 덧셈의 순서는 다수 별로 중요하지 않다. 부동 소숫점과 극한(saturation) 수학은 덧셈의 순서가 바뀌면 다른 답들을 낼 수 있지만 일반적인 wrap-around 정수 덧셈들은 덧셈의 순서에 관계없이 동일한 결과들을 낼 것이다. 그래서 우리는 이런 시퀀스를, 첫번째 두 값들 쌍들을 더하고, 그다음에 이런 부분합들을 더하고 이런식으로 단일 마지막 덧셈이 나올 때까지 계속하는, 세개의-구조화된 병렬 덧셈으로 재작성할 수 있다. 네개의 8-비트 값들의 벡터에 대해서 두 덧셈 단계들이 필요하다; 첫번째 단계는 두개의 8-비트 덧셈을 수행하고, 그다음 두개의 16-비트 결과 필드들을 생성한다(각각은 9-비트 결과를 담고 있다):

```
t = ((x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff));
```

두번째 스텝은 이런 두개의 9-비트 값들을 16-비트 필드들안에서, 단일 10-비트 결과를 만들기 위해, 더한다:

```
((t + (t >> 16)) & 0x000003ff)
```

실제, 두번째 스텝은 두개의 16-비트 필드 덧셈들을 수행한다... 그러나 머리 16-비트 덧셈은 의미가 없다. 이것이 바로 왜 결과가 단일 10-비트 결과 값에 대해서 마스킹되는가에 대한 이유이다.

"병렬 접두어(parallel prefix)" 연산으로 알려진 스캔은 다소 효율적으로 구현하기가 더 어렵다. 이것은 왜냐면, 축소(reduction)과 다르게, 스캔이 부분적인(partitioned) 결과를 내기 때문이다. 이런 이유로 스캔은 아주 명백한, 부분적인 연산들의 시퀀스를 사용해서 구현될 수 있다.

4.3 리눅스에서의 MMX SWAR

리눅스이 경우 IA32 프로세서들이 우리의 주요 관심사이다. AMD, Cyrix, 그리고 Intel 모두 동일한 MMX 명령어들을 구현한다고 하는 것은 굿뉴스이다. 그러나 MMX 성능은 서로 다르다; 예를 들어서 K6는 MMX 파이프라인을 단지 하나만 가진다 - (이에 반해서)Pentium with MMX는 두개를 가진다. Intel이 아직도 이런 멍청한 MMX 광고를 계속하고 있다는 것이 유일한 배드뉴스이다. ;-)

SWAR를 위하여 MMX를 사용하는 데는 실제 다음과 같은 세가지 접근법이 있다:

1. MMX 라이브러리 루틴들을 사용하는 것. 특별히 Intel은 몇가지 "성능 라이브러리들을(performance libraries)", <http://developer.intel.com/drg/tools/ad.htm> 개발했다. 이것은 일반 멀티미디어 작업들에 대해서 손으로-최적화된 다양한 루틴들을 제공한다. 적은 노력으로 많은 비-멀티미디어 알고리즘들이 대부분의 컴퓨터-집중 포션들의 일부가 이런 라이브러리 루틴들을 하나 또는 그이상 사용해서 구현될수 있도록 재작업될 수 있다. 이런 라이브러리들은 현재 리눅스에 대해서 사용불가능이지만 포팅 가능할 수 있다.

2. MMX 명령어들을 직접 사용하는 것. 이것은 다소 두가지 점들에 의해서 복잡하다. 첫번째 문제는 MMX가 프로세서에서 사용가능하지 않을 수 있어서 대체 구현물이 반드시 제공되어야 할수도 있다는 것이다. 두번째 문제는 리눅스에서 일반적으로 사용되는 IA32 어셈블러가 현재 MMX 명령어들을 이해하지 못한다는 것이다.

3. 적절한 MMX 명령어들을 직접 생성할 수 있는 고-수준 언어나 모듈 컴파일러(module compiler)를 사용하는 것. 그런 툴들은 현재 개발 중에 있지만 어떤 것도 아직 리눅스에서 완전한 기능을 가진 것이 없다. 예를 들어서 퍼듀 대학교 (<http://dynamo.ecn.purdue.edu/~hankd/SWAR/>)에서 우리는 현재 명시적으로 병렬 C 방언으로 작성된 함수들을 취해서 C 함수들로 가능한 SWAR 모듈들을 생성할 컴파일러를 개발하고 있지만, 아직 MMX를 포함해서 SWAR 지원이 가능한 것이면 무엇이든 사용한다. 첫번째 프로토타입 모듈 컴파일러는 1996년 가을에 만들어졌다. 그러나 이 기술을 사용가능한 상태까지 만드는 것은 처음에 예상한 것보다 더 오래 걸리고 있다.

요약하면 MMX SWAR는 여전히 사용하기에 어렵다. 그러나 여분의 노력을 조금 더하면 위에서 주어진 두 번째 접근법은 지금도 사용될 수 있다. 다음은 그 기본이다:

1. 프로세서가 MMX를 지원하지 않으면 MMX를 쓸 수 없다. 다음 GCC 코드는 MMX가 여러분의 프로세서에서 지원되는지 안되는지를 테스트하는 데 사용될 수 있다. 지원안되면 0이 리턴되고 지원되면 0이 아닌 값이 리턴된다.

```
inline extern
int mmx_init(void)
{
        int mmx_available;

        __asm__ __volatile__ (
                /* Get CPU version information */
                "movl $1, %%eax\n\t"
                "cpuid\n\t"
                "andl $0x800000, %%edx\n\t"
                "movl %%edx, %0"
                : "=q" (mmx_available)
                : /* no input */
        );
        return mmx_available;
}
```

2. MMX 레지스터는 기본적으로 GCC가 `unsigned long long`라고 부르는 것 중 하나를 갖고 있다. 그래서 이런 타입의 메모리-기반 변수들은 여러분의 MMX 모듈들과 그들을 호출하는 C 프로그램들간의 통신 메카니즘이 된다. 또는 MMX 데이터를 임의의 64-비트 정렬된 데이터 스트럭쳐로 선언할수도 있다 (여러분의 데이터 타입을 `unsigned long long` 필드를 가지는 `union`의 타입으로 선언함으로써 64-비트 정렬이 되도록 하는 것이 편리하다).

3. MMX가 사용가능이라면 여러분은, 각 명령을 인코드하는 `.byte` 어셈블리 지시어를 사용한 여러분의 MMX 코드를 작성할 수 있다. 예를 들어서 MMX 명령어 `PADDB MM0,MM1`는 다음과 같이 GCC 인-라인 어셈블리 코드로 인코딩될 수 있다:

```
__asm__ __volatile__ (".byte 0x0f, 0xfc, 0xc1\n\t");
```

   MMX는 부동 소숫점 연산들에 대해서 사용되는 하드웨어와 동일한 것들을 사용한다는 것을 기억하자. 그래서 MMX 코드와 서로 섞인 코드는 부동 소숫점 연산들을 호출해서는 안된다. 부동 소숫점 스택도 또한 MMX 코드를 실행하기 전에 비워져야 한다; 부동 소숫점 스택은 일반적으로 부동 소숫점을 사용하지 않는 C 함수의 시작점에서 비워진다.

4. 다음과 같이 코딩될 수 있는 것처럼, `EMMS` 명령을 실행함으로써, MMX 코드를 종료하자:

```
__asm__ __volatile__ (".byte 0x0f, 0x77\n\t");
```

위의 것이 아주 이상하고 조잡하게 보인다면 그렇다. 그러나 MMX는 여전히 꽤 젊다... 이 문서의 나중 버전은 MMX SWAR를 프로그램하는 좀 더 나은 방법들을 제공할 것이다.

제 5 절 리눅스가 호스트하는 부속 프로세서(Linux-Hosted Attached Processors)

이런 접근은 요새 별로 인기가 없지만 다른 병렬 처리 방법들이 리눅스 시스템을 호스트에 부속 병령 컴퓨팅 시스템으로 사용함으로써 낮은 비용에 고성능을 얻는 것은 거의 불가능하다. 문제는 소프트웨어 지원이 아주 작다는 것이다; 여러분은 거의 혼자이다.

5.1 리눅스 PC는 좋은 호스트이다(A Linux PC Is A Good Host)

일반적으로 부속 병렬 프로세서들은 특정 타입의 기능들을 수행하는 데 전문화되는 경향이 있다.

여러분이 어쩌면 혼자일런지 모른다는 사실에 기죽기 전에 다음과 같은 것을 이해하는 것은 유용하다. 즉, 리눅스 PC가 적절하게 특정 시스템을 호스트하도록 하는 것은 어려울 수 있을지라도 리눅스 PC는 이런 타입으로 사용되는 데에는 적절한 몇개 안되는 플랫폼들 중 하나이다.

PC들은 두가지 주요한 이유 때문에 좋은 호스트이다. 첫번째는 싸고 쉬운 확장 능력이다; 더 많은 메모리, 디스크, 네트웍 등과 가은 리소스들이 쉽게 PC에 추가된다. 두번째는 인터페이스의 용이성이다. ISA와 PCI 버스 프로토타입 카드들이 널리 사용가능할뿐만 아니라 병렬 포트는 완전히 비-침략적인 인터페이스로 적당한 성능을 제공한다. IA32 분리된 I/O 스페이스는 또한 개별 I/O 포트 주소들의 레벨에서 하드웨어 I/O 주소 프로텍션을 제공함으로써 인터페이스를 용이하게 한다.

리눅스는 또한 좋은 호스트 OS이다. 전체 소스 코드의 자유로운 사용 가능성, 많은 "핵킹" 카이드들, 이들은 명백히 대단한 도움이다. 그러나 리눅스는 또한 괜찮은 거의-실-시간 스케줄링을 제공하고 `<http://luz.cs.nmt.edu/~rtlinux/>`에는 리눅스의 진정한 실-시간 버전조차 있다. 아마도 완전한 UNIX 환경을 지원하는 반면 리눅스는 Microsoft DOS 또는 Windows에서 실행할 수 있도록 작성된 개발 툴들을 지원하는 것이 조금 더 중요한 사실이다. MSDOS 프로그램들은, 글자 그대로 MSDOS를 실행할 수 있는 프로텍티드 가상 머쉰을 제공하는, `dosemu`를 사용한 리눅스 프로세스 안에서 실행될 수 있다. 리눅스는 좀 더 직접적으로 Windows 3.xx 프로그램들에 대해서 지원한다: `wine`, `<http://www.linpro.no/wine/>`, 과 같은 자유 소프트웨어는 UNIX/X 환경안에서 정확하고 효율적으로 대부분의 프로그램들을 실행할만큼 충분히 잘 Windows 3.11을 시뮬레이트한다.

다음 두 섹션들은 내가 리눅스에서 지원되었으면 하고 바라는 부속 병렬 시스템들에 대한 예제들을 제공한다....

5.2 그것에 DSP를 적용했는가(Did You DSP That)?

고-성능 DSP(디지털 시그널 처리(Digital Signal Processing)) 프로세서 시장이 번성중이다. 비록 이런 칩들이 일반적으로 어플리케이션-종속적인 시스템들에 임베딩되도록 고안된 것이지만, 그들은 또한 거대한 부속 병렬 컴퓨터들 또한 만들고 있다. 왜 그런가?

- Texas Instruments (`<http://www.ti.com/>`) TMS320와 Analog Devices (`<http://www.analog.com/>`) SHARC DSP 패밀리와 같은 많은 것들이 "접착(glue)" 로직이 거의 없는 또는 전혀 없는 병렬 기계들을 만들도록 고된 것이다.
- 이들은 아주, 특별히 MIP나 MFLOP 당 비용이, 싸다. 기본 지원 로직의 비용을 포함해서 DSP 프로세서가 비교가능한 성능을 가지는 PC 프로세서의 비용의 10분의 1이라고 한다.
- 그들은 많은 전력을 쓰거나 많은 열을 발생하거나 하지 않는다. 이것은 전통적인 피씨의 파워 서플라이에 의한 전력을 이런 일련의 칩들에 공급하는 것이 가능하다는 것을 의미한다 - 그리고 그것들을 여러분의 피씨 케이스에 넣어도 이것이 오븐이 되지 않을것이라는 것을 의미한다.
- 고-수준 (예, C) 컴파일러들이 잘 사용할 것같지 않은 대부분의 DSP 명령어 집합 - 예를 들어서 "비트 역방향 어드레싱(

- 고-수준 (예, C) 컴파일러들이 잘 사용할 것같지 않은 대부분의 DSP 명령어 집합에는 이상하게 보이는 것들이 있다 - 예를 들어서 "비트 역방향 어드레싱(Bit Reverse Addressing)". 부속 병렬 시스템을 사용하면 그런 호스트에서 대부분의 코드를 직선적으로(straightforwardly) 컴파일하고 실행하는 것이 가능하다. 이에 반해서 DSP 위에서 시간을 대부분 잡아먹는 몇개 안되는 알고리즘들은 조심스럽게 손으로-튜닝된 코드로 실행된다.
- 이런 DSP 프로세서들은 실제 UNIX-like OS에서 실행되도록 고안된 것이 아니고 일반적으로 독립-실행형 범용 컴퓨터 프로세서들과 마찬가지로 좋지 않다. 예를 들어서 많은 것들이 메모리 관리 하드웨어를 가지고 있지 않다. 다른 말로 하면 그들은 좀 더 범용 기계들에 의해서 호스트되어야 가장 잘 작동된다... 리눅스와 같은.

어떤 오디오 카드들과 모뎀들은 리눅스 드라이버들이 억세스할 수 있는 DSP 프로세서들을 포함하고 있지만 네개 또는 그 이상의 DSP 프로세서들을 가지는 부속 병렬 시스템을 사용하면 그 댓가가 크다.

Texas Instruments TMS320 시리즈, `<http://www.ti.com/sc/docs/dsps/dsphome.htm>`, 는 아주 오랫동안 아주 인기가 있었고 TMS320-기반 병렬 프로세서를 만들기가 쉬웠기 때문에 사용가능한 그런 시스템들이 꽤 있었다. TMS320에는 정수-만의 버전과 부동-소숫점 가능 버전들이 있다; 더 오래된 디자인들은 다소 비일상적인 단일-정밀도 부동-소숫점 포멧을 사용했지만 새로운 모델들은 IEEE 포멧들을 지원한다. 오래된 TMS320C4x ('C4x 로 알려짐)는 TI-종속적인 단일-정밀도 부동-소숫점 포멧을 사용해서 80 MFLOPS 까지 획득했다; 이에 반해서 단일 'C67x 은 VelociTI라고 불리는 VLIW-기반 칩 아키텍쳐를 사용해서, 1 GFLOPS 단일-정밀도 또는 IEEE 부동 소수점 연산에 대해서 420 MFLOPS 배-정밀도까지 제공할 것이다. 멀티프로세서로 이런 칩들의 그룹을 설정하는 것이 쉬울뿐 아니라 단일 칩안에서도 'C8x 멀티프로세서는 두개 또는 네개의 정수 부속 DSP들과 함께 100 MFLOPS IEEE 부동-소숫점 RISC 마스터 프로세서를 제공할 것이다.

몇개의 부속 병렬 시스템들보다 더 많이 사용된바 있는 다른 DSP 프로세서 패밀리는 Analog Devices `<http://www.analog.com/>` 사의 SHARC(ADSP-2106x로 알려짐)이다. 이런 칩들은 외부 접착(glue) 논리 없이 6개의 프로세서 공유 메모리 멀티프로세서로 설정될 수 있다. 그리고 점 더 큰 시스템들도 여섯개의 4-비트 links/chip(칩당 링크)를 사용해서 설정될 수 있다. 대부분의 더 큰 시스템들은 군사용 어플리케이션을 목표로 하는 것 같고 약간 비싸다. 그러나 Integrated Computing Engines, Inc., `<http://www.iced.com/>`, 회사는 GreenICE라고 불리는 흥미로운 조그만 두-보드 PCI 카드 셋을 만들었다. 이 유닛은 16개의 SHARC 프로세서들 배열을 가지고 있고 단일-정밀도 IEEE 포멧을 사용해서 약 1.9 GFLOPS의 최고 속도를 낼 수 있다. GreenICE는 $5,000 미만의 가격이다.

내 의견으로는 부속 병령 DSP들은 실제로 리눅스 병렬 처리 커뮤니티가 더 많은 신경을 써야 마땅할 것이라고 생각한다....

5.3 FPGAs과 재설정 가능한 논리 연산

병렬 처리가 가장 높은 성능향상을 얻기 위한 것이 전부이라면 왜 커스텀 하드웨어를 만들지 않는가? 글쎄, 우리는 모두 답을 알고 있다; 이것은 너무 비싸며 개발하기에 시간이 너무 오래 걸리고 조금이라도 알고리즘을 변경할 때면 쓸모없는 것이 되버린다. 기타 등등. 그러나 전자적으로 재프로그래밍 가능한 FPGA(필드 프로그래머블 게이트 어레이(Field Programmable Gate Arrays))들의 요즘의 진보가 이런 제약들의 대부분을 무력화시켜 버렸다. 지금 게이트 밀집도가 충분히 높아서 단순한 전체 프로세서가 하나의 FPGA에 들어가도록 만들어질 수 있고 FPGA를 재설정(재프로그램)하는 것도 또한, 한 알고리즘의 한 국면에서 다음으로 옮겨갈 때라도 재설정하는 것이 타당할만큼의 수준까지 낮아졌다.

이 내용은 심장이 약한 사람들을 위한 것이 아니다: 여러분은 FPGA 설정에 대해서, 리눅스 호스트 시스템 위의 프로그램들에 대해서 인터페이스하는 로우-레벨 코드를 작성하는 일과 함께, VHDL과 같은 하드웨어 기술(description) 언어들로 작업해야 한다. 그러나 FPGA의 비용은 낮고 특별히 낮은-정밀도 정수 데이터(실제, 이런 재료의 조그만 상위집합에 대해서는 SWAR가 더 낫다)에 대해서 작업하는 알고리즘들에 대해서 비용이 낮고, FPGA는 여러분이 데이터를 제공하는 속도만큼 빠르게 복잡한 연산들을 수행할 수

있다. 예를 들어서 단순한 FPGA-기반 시스템들은 유전자 데이터베이스 검색에서 슈퍼컴퓨터보다 더 나은 속도를 만든다.

적절한 FPGA-기반 하드웨어를 만드는 다른 회사들이 있지만 다음과 같은 두 회사가 좋은 샘플을 제시한다.

Virtual Computer Company는 동적으로 재설정 가능한 SRAM-기반 Xilinx FPGA들을 사용한 다양한 제품들을 제공한다. 그들의 8/16비트 "가상 ISA 프로토 보드(Virtual ISA Proto Board)" `<http://www.vcc.com/products/isa.html>`는 $2,000 미만이다.

알테라(Altera) ARC-PCI(Altera Reconfigurable Computer, PCI bus), `<http://www.altera.com/html/new/pressrel/pr_arc-pci.html>`, 는 비슷한 타입의 카드이지만 알테라 FPGA들과 ISA가 아닌 PCI 버스 인터페이스를 사용한다.

많은 설계 툴들, 하드웨어 기술(description)언어, 컴파일러, 라우터, 맵퍼 등은 윈도우즈나 DOS에서만 실행되는 오브젝트 코드로 제공된다. 호스트 피씨에다 DOS/Windows를 가진 디스크 파티션을 가지고 그것들이 필요할 때마다 리부팅한다. 그러나 이들 소프트웨어 패키지들은 리눅스에서 `dosemu`를 사용해서 또는, `wine`와 같은 윈도우즈 에뮬레이터를 사용해서 실행될 수 있다.

제 6 절 일반적인 관심거리 중에서

이 섹션에서 다루어지는 내용은 모든 리눅스의 네가지 병렬 처리 모델들에 적용되는 것이다.

6.1 프로그래밍 언어와 컴파일러

나는 주로 컴파일러 연구자로 알려져 있다. 그래서 나는 리눅스 시스템들에 대한 효율적인 병렬 코드를 자동으로 생성하는 위대한 컴파일러들이 많이 있다고 말할 수 있었으면 한다. 불행하게도 다양한 명시적 통신과 다른 병렬 연산들을 사용한 여러분의 병렬 프로그램을 GCC로 컴파일될 C 코드로 표현함으로써 성능을 높일 수 있다는 것은 어렵다는 것이 진실이다.

다음 언어/컴파일러 프로젝트들이 고-수준 언어들로부터 효율적인 코드를 만드는 노력을 하고 있다. 일반적으로 이들 각각은 그것이 지향하는 프로그래밍 작업들의 종류에 대해서는 효율적이지만 어떤 것도 강력한 범용 언어이지 않고, GCC로 컴파일할 C 프로그램을 작성하는 것을 영원히 멈추게 할 만한 것이 없다. (그래도) 이것은 좋다. 이런 언어들과 컴파일러들을 그들이 의도된 바대로 사용하고 여러분은 더 짧은 개발 시간, 쉬운 디버깅과 유지 보수 등의 보상을 받을 것이다.

여기에 리스트된(알파벳 순서로) 것들 외에 많은 언어들과 컴파일러들이 있다. 자유롭게 사용가능한 컴파일러들(이들중 대부분은 리눅스 병렬 처리에 대해서 아무것도 하지 못한다)은 `<http://www.idiom.com/free-compilers/>`에 있다.

6.1.1 Fortran 66/77/PCF/90/HPF/95

적어도 과학 컴퓨팅 사회에서는 언제나 포트란(Fortran)이 있다. 물론 이제 포트란은 1966년 ANSI 표준에서 그랬던 것과 똑같은 것을 의미하지 않는다. 기본적으로 포트란 66은 아주 단순한 것이다. 포트란 77은 많은 것들을 추가했고 이들 중 가장 주목할 만한 것은 문자 데이터에 대한 개선된 지원과 `DO` 루프 문법의 변경이다. PCF (Parallel Computing Forum) 포트란은 77에다 다양한 병렬 처리 지원을 더하려고 시도하였다. 포트란 90은 완전한-기능의 현대 언어이다. 이것은 기본적으로 C++-비슷한 객체-지향 프로그래밍 기능들과 병렬 배열 문법을 77 언어에 추가한 것이다. 두 버전들(HPF-1과 HPF-2)를 가지는 HPF (High-Performance Fortran, `<http://www.crpc.rice.edu/HPFF/home.html>`)는 기본적으로 진보된, 표준화된, 우리들이 보통 CM 포트란, MasPar 포트란, 또는 포트란 D로 알고 있는 것의 버전이다; 이것은 다양한 병렬 처리 개선점들을 넣어서 포트란 90을 확장한 것이다. 주로 데이터 레이아웃을 지정하는 데 촛점을 둔 것이다. 마지막으로 포트란 95는 90을 상대적으로 적게 증진시키고 개선한 것이다.

C로 작업한 것은 일반적으로 f2c, g77 (리눅스-종속적인 훌륭한 개관은 <http://linux.uni-regensburg.de/psi_linux/gcc/html_g77/g77_91.html>에 있다), 또는 <http://extweb.nag.co.uk/nagware/NCNJNKNM.html> 상업용 버전 포트란 90/95과도 잘 작동한다. 이것은 이들 모든 컴파일러들이 결과적으로 GCC의 백-엔드에서 사용된 것과 코드-생성에서 동일하기 때문이다.

SMP에 대한 코드를 생성할 수 있는 상업용 포트란 병렬기(parallelizer)들은 <http://www.kai.com/>과 <http://www.psrv.com/vast/vast_parallel.html>에서 찾을 수 있다. 이런 컴파일러들이 SMP 리눅스에서 작동할런지 안할런지는 모르지만 표준 POSIX 스레드들(즉, LinuxThreads)가 SMP 리눅스하에서 작동한다면 그것은 가능할 것이다.

포트랜드 그룹(Portland Group)은, <http://www.pgroup.com/>, SMP 리눅스에 대한 코드를 생성하는 상업용 병렬화 HPF 포트란(그리고 C, C++)를 가지고 있다; 그들은 또한 MPI나 PVM을 사용한 클러스터들을 타겟으로 한 버전도 갖고 있다. <http://www.apri.com/>의 FORGE/spf/xHPF 제품들도 SMP들이나 클러스터들에 대해서 유용할 것이다.

병렬 리눅스 시스템들과 작업이 가능하도록 만들어질 수 있는, 자유롭게 사용가능한 병렬화 포트란들은 다음과 같은 것들을 포함한다:

- ADAPTOR (자동 데이터 병렬화 변환기(Automatic DAta Parallelism TranslaTOR), <http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html>), 는 HPF를 MPI 또는 PVM 호출들로 이루어진 포트란 77/90 코드로 변환할 수 있다. 그러나 리눅스는 언급하지 않는다.
- 카네기 멜론의 Fx <http://www.cs.cmu.edu/~fx/Fx> 는 몇가지 워크스테이션 클러스터들을 목표로 삼는다. 그러나 리눅스는?
- HPFC (프로토타입 HPF 컴파일러, <http://www.cri.ensmp.fr/~coelho/hpfc.html>) PVM 호출들로 이루어진 포트란 77을 생성한다. 이것은 리눅스 클러스터에서 사용가능인가?
- PARADIGM (분산-메모리 범용 멀티컴퓨터에 대한 병렬화 컴파일러(PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers), <http://www.crhc.uiuc.edu/Paradigm/>) 는 리눅스에서 사용될 수 있는가?
- Polaris 컴파일러, <http://ece.www.ecn.purdue.edu/~eigenman/polaris/>, 는 공유 메모리 멀티프로세스들에 대한 포트란 코드를 생성하고 얼마 안있어서 PAPERS 리눅스 클러스터를 다시 목표로 삼을 것이다.
- PREPARE, <http://www.irisa.fr/EXTERNE/projet/pampa/PREPARE/prepare.html>, 는 MPI 클러스터들을 목표로 삼는다... IA32 프로세서들에서 실행되는 코드를 생성할 수 있는지 없는지는 분명하지 않다.
- ADAPT와 ADLIB를 조합해서, shpf(고성능 포트란 컴파일 시스템 부분집합(Subset High Performance Fortran compilation system), <http://www.ccg.ecs.soton.ac.uk/Projects/shpf/shpf.html>)는 MPI 호출들을 가지는 포트란 90을 생성하는 퍼블릭 도메인에 있는 것이다... 그래서 여러분이 리눅스에서 포트란 90 컴파일러를 가지고 있다면...
- SUIF (스탠포드 대학교 중간 형식(Stanford University Intermediate Form), <http://suif.stanford.edu/> 참조) 는 C와 포트란 모두에 대한 병렬화 컴파일러를 가지고 있다. 이것은 전미 컴파일러 인프라 프로젝트(National Compiler Infrastructure Project)에 대해서도 관심을 가지고 있다. 그래서 병렬 리눅스 시스템들을 목표로 하는 사람이 있는가?

포트란의 다양한 방언들에 대한 잠재적으로 유용한 많은 컴파일러들을 빼먹었다고 확신하지만 추적하기에 어려운 점이 많다. 나중에 나는 리눅스에서 작동한다고 알려진 그런 컴파일러들만 모아 보고자 한다. pplinux@ecn.purdue.edu로 비평이나 교정을 이메일로 보내주기 바란다.

6.1.2 GLU (Granular Lucid)

GLU (Granular Lucid) 는 집약적이고(intensional) (Lucid) 긴급한 모델들을 조합한 하이브리드 프로그래밍 모델에 기반한 고-수준 프로그래밍 시스템이다. 이것은 PVM과 TCP 소켓들을 지원한다. 이것은 리눅스에서 실행되는가? 좀 더 많은 정보가 `<http://www.csl.sri.com/GLU.html>`에 있다.

6.1.3 Jade와 SAM

Jade 는 C를 확장해서 순차적이고 긴급한(imperative) 프로그램들 안에서 성긴 협력을 개발하도록 한 병렬 프로그래밍 언어이다. 이것은 분산 공유 메모리 모델을 가정한다. 이것은 PVM을 사용하는 워크스테이션 클러스터들에 대해서 SAM에 의해 구현된다. 좀 더 많은 정보는 `<http://suif.stanford.edu/~scales/sam.html>`에서 찾을 수 있다.

6.1.4 Mentat과 Legion

Mentat는 워크스테이션 클러스터들에서 작동하고 리눅스로 포팅된바 있는 객체-지향 병렬 처리 시스템이다. Mentat 프로그래밍 언어(MPL)은 C++에 기반하고 있는 객체-지향 프로그래밍 언어이다. Mentat 실시간 시스템은 비-블록킹 RPC들과 약간 닮은 어떤 것들을 사용한다. 좀 더 많은 정보는 `<http://www.cs.virginia.edu/~mentat/>`에서 찾을 수 있다.

Legion `<http://www.cs.virginia.edu/~legion/>` 는 Mentat 의 상위에 만들어진 것이고 WAN으로 묶인 기계들에 대해서 단일 가상 기계를 제공한다.

6.1.5 MPL (MasPar 프로그래밍 언어)

Mentat 의 MPL과 혼동되지 말자. 이 언어는 원래 MasPar SIMD 슈퍼컴퓨터를 위한 원시 병렬 C 방언으로써 개발되었다. 글쎄, MasPar는 실제 더이상 장사를 하지 않지만(그들은 이제 NeoVista Solutions, `<http://www.neovista.com>`, 데이터 마이닝 회사이다), 그들의 MPL 컴파일러는 GCC를 사용해서 개발되었었다. 그래서 이것은 아직도 자유롭게 사용가능하다. Huntsville의 알라바마 대학교와 퍼듀 대학교 이들과 조인트해서 MasPar 의 MPL은 AFAP 호출들을 가진 C 코드를 생성하는 것으로 목표가 바뀌었다(섹션 3.6을 참조), 그래서 리눅스 SMP 및 클러스터들 위해서 작동한다. 그러나 그 컴파일러는 다소 버그가 많다. `<http://www.math.luc.edu/~laufer/mspls/papers/cohen.ps>` 참조.

6.1.6 PAMS (병렬 어플리케이션 관리 시스템(Parallel Application Management System))

Myrias 는 PAMS (병렬 어플리케이션 관리 시스템) 이라고 불리는 소프트웨어 제품을 파는 회사이다. PAMS 는 가상 공유 메모리 병렬 처리를 위한 아주 단순한 지시어들을 제공한다. 리눅스 기계들의 네트웍은 아직 지원되지 않는다. 좀 더 자세한 정보를 위해서는 `<http://www.myrias.com/>`를 참조하자.

6.1.7 Parallaxis-III

Parallaxis-III 는 데이터 병렬화 (SIMD 모델)에 대한 "가상 프로세서와 커넥션(virtual processors and connections)"으로 Modula-2를 확장한 구조적 프로그래밍 언어이다. Parallaxis 소프트웨어는 순차 및 병렬 컴퓨터 시스템들을 위한 컴파일러들과, 디버거(gdb와 xgdb 디버거에 대한 확장판), 그리고 서로 다른 영역의, 특별히 이미지 처리 영역의, 많은 양의 샘플 알고리즘들로 이루어져 있다. 이것은 순차 리눅스 시스템들 위해서 실행된다... 구 버전은 다양한 병렬 타겟들을 지원했었다. 그리고 새로운 버전도 (예, PVM 클러스터를 타겟으로 삼음) 또한 그럴것이다. 좀 더 자세한 정보는 `<http://www.informatik.uni-stuttgart.de/ipvr/bv/p3/p3.html>`에서 찾을 수 있다.

6.1.8 pC++/Sage++

pC++/Sage++ 는 어떤 기본 "요소(element)" 클래스로부터 "객체들의 집합체(collections of objects)"를 사용하여 데이터-병렬 스타일 작업들을 허용하는 C++에 대한 언어 확장이다. 이것은 PVM에서 실행될 수 잇는 C++ 코드를 생성하는 선행 프로세서이다. 이것은 리눅스에서 작동하는가? 좀 더 많은 정보는 <http://www.extreme.indiana.edu/sage/>에서 찾을 수 있다.

6.1.9 SR (리소스 동기(Synchronizing Resources))

SR (리소스 동기(Synchronizing Resources))는 리소스들이 그들이 공유하는 프로세스들과 변수들을 캡슐화하는 협력(concurrent) 프로그래밍 언어이다; 연산(operation)들이 프로세스 상호작용에 대한 주요 메카니즘을 제공한다. SR 은 호출과 서비스 작업들에 대한 메카니즘들의 새로운 정합을 제공한다. 결과적으로 모든 로컬 그리고 리모트 프로시저 호출, 랑데뷰, 메시지 전달, 동적 프로세스 생성, 멀티캐스트, 그리고 세마포어들이 지원된다. SR 은 또한 공유된(shared) 전역 변수들과 연산(operation)들을 지원한다.

이것은 리눅스로 포팅되었지만 그것으로 실행될 수 있는 병렬화가 무엇인지는 분명하지 않다. 좀 더 많은 정보는 <http://www.cs.arizona.edu/sr/www/index.html>에서 가능하다.

6.1.10 ZPL과 IronMan

ZPL 은 공학과 과학 어플리케이션들을 지원하도록 고안된 배열-기반 프로그래밍 언어이다. 이것은 IronMan 이라고 불리는 단순한 메시지-전달 인터페이스에 대한 호출을 생성하며 이런 인터페이스를 구성하는 몇가지 함수들이 거의 모든 메시지-전달 시스템을 사용해서 쉽게 구현될 수 있다. 그러나 워크스테이션 클러스터들 상의 PVM 과 MPI, 그리고 리눅스가 지원되는 것이 주요 타겟이다. 좀 더 자세한 정보는 <http://www.cs.washington.edu/research/projects/orca3/zpl/www/>에서 찾을 수 있다.

6.2 성능 문제(Performance Issues)

특정한 마더보드, 네트웍 카드들 등을 어떤 것이 최고인가를 알아보려고 벤치마킹하는 데, 많은 시간을 소비하는 사람들이 많이 있다. 이런 것의 문제는 여러분이 어떤 것을 벤치마킹할 수 있을 때에는 이미 그것이 사용할 수 있는 것 중에 최고가 더이상 아니라는 것이다; 이것은 심지어 시장에서 사라지고 완전 다른 속성들을 가진 개선된 모델로 교체되었을수 있다.

PC 하드웨어를 사는 것은 오렌지 쥬스를 사는 것과 같다. 보통 이것은 라벨에 붙은 회사 이름이 무엇인지는 상관이 없고 아주 좋은 재로로 만들어진다. 구성 성분들(또는 오렌지 쥬스 농축액)이 무엇으로 만들어졌는지 신경쓰거나 아는 사람은 거의 없다. 즉, 여러분이 신경쓸 하드웨어 차이들은 별로 안된다. 여러분이 리눅스로 쓸 하드웨어에 대해서 기해할 수 있는 것이 무엇인가를 확실히 알고 빠른 배달, 좋은 가격, 그리고 반품에 대한 적절한 정책에 대해서만 신경을 집중하라는 것이 내 조언이다.

서로 다른 PC 프로세서들에 대한 훌륭한 개관은 <http://www.pcguide.com/ref/cpu/fam/>에 있다; 사실 완전한 WWW 사이트 <http://www.pcguide.com/>에는 PC 하드웨어의 좋은 기술적인 개관들이 모여 있다. 특정 하드웨어 설정들의 성능에 대해서 조금 아는 것도 유용하다. 그리고 리눅스 벤치마킹 HOWTO(Linux Benchmarking HOWTO) <http://sunsite.unc.edu/LDP/HOWTO/Benchmarking-HOWTO.html>가 시작하기에 좋은 곳이다.

인텔 IA32 프로세서들은 실행중인 시스템의 성능을 정교한 세부사항까지 측정하는 데 사용되는 많은 특수한 레지스터들을 가진다. 인텔 VTune, <http://developer.intel.com/design/perftool/vtune/>, 는 아주 완전한 코드-튜팅 시스템에서 성능 레지스터들을 넓게 사용한다... 이것은 불행하게도 리눅스에서 실행되지 않는다. 펜티엄 성능 레지스터들을 억세스하기 위한 로딩 가능한 모듈 장치 드라이버, 그리고 라이브러리 루틴들은 <http://www.cs.umd.edu/users/akinlar/driver.html>에서 찾을 수 있다. 이런 성능 레지스터들은 다른 IA32 프로세서들과 다르다는 것을 기억하자; 이런 코드는 486, Pentium Pro, Pentium II, K6 등과는 작동하지 않고 Pentium에 대해서만 작동한다.

성능에 대한 다른 언급은, 커다란 클러스터들을 만들고 그것을 조그만 공간에 넣고자 하는 사람들에 특별히, 적절하다. 적어도 요즘의 어떤 프로세서들은 온도 센서를 내장하고 있고 운영 온도가 너무 높을 때 내부 클럭을 늦추는 데 사용되는 회로들(열 생성을 줄이고 신뢰도를 높이는 시도)을 가지고 있다. 나는 모든 사람들이 펠티에[2] 장치(열 펌프)를 사서 각 CPU를 식힐 필요는 없다고 제안하지만 높은 운영 열이 성분들의 사용시간을 줄일뿐만 아니라 - 시스템 성능을 직접 줄일 수도 있다는 것을 알아야 한다. 여러분의 컴퓨터들을 공기흐름을 차단하는 물리적 배치속에 놓지 말고 제한된 영역안에서 열을 잡아라. 기타 등등.

마지막으로 성능이란 단순히 속도뿐만이 아니고 신뢰도와 가용성도 포함된다. 높은 신뢰도란 여러분의 시스템이, 비록 구성 요소들이 실패하더라도, 거의 절대로 죽지(crash) 않는다는 것을 의미한다... 이것은 일반적으로 여분의 파워 공급기와 핫-스왑 마더보드와 같은 특수 기능들을 요구한다. 이것은 보통 싸지 않다. 높은 가용성이란 거의 모든 시간에 사용 가능하다는 개념을 말한다... 시스템은 그 구성요소가 실패할 때 죽을 수도 있지만 시스템은 재빨리 고쳐지고 리부팅된다. 많은 기본 이슈들을 논의한 High-Availability HOWTO가 있다. 그러나 클러스터에 대해서 특별히, 높은 가용성은 몇개의 여분을 가지는 것으로 쉽게 획득될 수 있다. 나는 적어도 한개의 여분을 권장하는 바이고 커다란 클러스터에서 16개 기계마다 하나씩의 여분을 적어도 가지는 것을 선호한다. 잘못된 하드웨어를 버리고 그것을 여분의 것으로 교체하는 것은 유지보수 계약보다도 더 높은 가용성과 더 낮은 비용을 얻을수 있도록 한다.

6.3 결론 - 거기에 있다.

그래 리눅스를 사용해서 병렬 처리를 하는 사람이 있는가? 그렇다!

많은 사람들이 많은 병렬-처리 슈퍼컴퓨터 회사들이 죽는다는 것이 병렬 처리가 시들시들해지기 시작한다는 것을 의미하는 것이 아닌가하고 생각했던 것은 아주 오래 전의 일이 아니다. 나는 그당시 그럴 것이라고 생각하지 않았고(내가 실제로 일어날 것이라고 생각한 것에 대한 것을 재미거리로 보려면 `<http://dynamo.ecn.purdue.edu/~hankd/Opinions/pardead.html>`를 참조하기 바란다), 병렬 처리가 지금 또 다시 일어서고 있는 것은 명백한 사실이다. 심지어 얼마전에 병렬 슈퍼컴퓨터를 만드는 것을 중지한 Intel이 MMX와 앞으로 나올 IA64 EPIC (Explicitly Parallel Instruction Computer)과 같은 것들에서 병렬 처리 지원을 한다고 자랑하고 있다.

여러분이 선호하는 검색 엔진에서 "Linux"와 "paralle'을 검색한다면 여러분은 몇개의 사이트들이 리눅스를 사용한 병렬 처리에 포함된다는 것을 알게 될 것이다. 특별히 Linux PC 클러스터들은 모든 곳에서 떠오르는 것처럼 보일 것이다. PC 하드웨어의 저비용 고성능과 합쳐서 리눅스의 적절함(appropriateness)는 리눅스를 사용한 병렬 처리를 작고, 예산이 적은 그룹들과 크고 예산이 많고 국가적인 연구소들 모두의 인기있는 슈퍼컴퓨팅 접근으로 만들었다.

이 문서 도처에 있는 다양한 프로젝트들이 비슷한 병렬 리눅스 설정을 가지는 "유사한" 연구 사이트들 리스트를 관리한다. 그러나 `<http://yara.ecn.purdue.edu/~pplinux/Sites/>`에, 병렬 처리에 리눅스 시스템들을 사용하는 다양한 사이트들의 사진, 설명, 그리고 연락 정보들을 제공하기 위해서 고안된 문서들이 있다.

- 여러분은 반드시 "영속적인" 병렬 리눅스 사이트를 가져야 한다: SMP, 기계들의 클러스터, SWAR 시스템, 또는 부속 프로세서를 가지는 PC 등등은 사용자들에게 *리눅스에서 병렬 프로그램들을 수행할 수 있도록* 설정된다. 병렬 처리를 직접 지원하는 리눅스-기반 소프트웨어 환경 (예, PVM, MPI, AFAPI)이 그 시스템에 반드시 설치되어야 한다. 그러나, 하드웨어는 리눅스의 병렬 처리에 매일 필요가 없고 병렬 프로그램들이 실행중이 아닐 때 완전히 다른 목적들을 위해서 사용될 수 있다.

- 여러분의 사이트가 리스트되도록 요구하자. 여러분의 사이트 정보를 `pplinux@ecn.purdue.edu`에게 보내자. 여러분의 사이트 정보에 대해서 다른 엔트리들에서 사용된 포멧을 따르기 바란다. *그 사이트에 대한 접촉 가능한 사람으로부터의 명시된 요구가 없으면 어떤 사이트도 리스트에 추가되지 않을 것이다.*

[2]역자주: 펠티에 효과 - 이종(異種)의 금속 접촉면에 약한 전류가 흘렀을 때 열이 발생 또는 흡수되는 현상

현재 리스트에는 14개의 클러스터들이 있지만 우리는 적어도 수십개의 클러스트들이 전세계에 있다는 것을 안다. 물론 리스트가 암시나 보증, 기타 등등을 의미하지 않는다; 우리는 단지 리눅스를 사용한 병렬 처리를 포함한 인식, 연구, 그리고 협력을 증진하기를 바랄뿐이다.