Beowulf HOWTO

Jacek Radajewski 와 Douglas Eadline v1.1.1, 22 November 1998

역자 : 박재성, *ospace@chollian.net* <mailto:ospace@chollian.net>, *kida@inet.cheju.ac.kr* <mailto:kida@inet.cheju.ac.kr>
일시 : 1999년 5월 ??일

이 문서는 베오울프 슈퍼컴퓨터 구조를 소개하고 병렬 프로그램밍에서의 기본적인 지식을 제공하는 것 외에 다른 특정 문서와 웹페이지로의 링크도 포함하고 있다.

목 차

제 1 절 머리말 **2**

1.1 Disclaimer . . . 2

1.2 판권 . . . 2

1.3 이 문서에 대해 . . . 2

1.4 저자들에 관해 . . . 3

1.5 감사의 말 . . . 3

제 2 절 개요 **3**

2.1 누가 이 HOWTO 문서를 읽어야 하는가? . . . 3

2.2 베오울프란? . . . 3

2.3 분류 . . . 4

제 3 절 구조 개요 **5**

3.1 무엇 처럼 보이는가? . . . 5

3.2 다른 노드를 어떻게 이용할 것인가? . . . 5

3.3 베오울프가 COW와 어떻게 다른가? . . . 6

제 4 절 시스템 디자인 **7**

4.1 병렬 계산에있어서 배경 요약. . . . 7

4.2 병렬 계산 방법 . . . 7

4.2.1 왜 한 개 이상의 CPU가 필요한가? . . . 7

4.2.2 병렬 처리 가게 . . . 8

4.3 병렬 계산을 위한 구조 . . . 9

4.3.1 하드웨어 구조 . . . 9

4.3.2 소프트웨어 API 구조 . . . 9

4.3.3 풀그림 구조 . . . 10

4.4 적합성 . . . 10

4.5 병렬 소프트웨어 작성과 포팅 11
4.5.1 프로그램의 병행 부분 결정 12
4.5.2 병렬 효율 계산 12
4.5.3 프로그램에서 병행 부분을 기술 12

제 5 절 베오울프 자원들 13
5.1 시작점 13
5.2 문서 13
5.3 논문 14
5.4 소프트웨어 14
5.5 베오울프 머신들 14
5.6 다른 재미있는 사이트 15
5.7 역사 15

제 6 절 밑그림 15
6.1 sum.c 15
6.2 sigmasqrt.c 15
6.3 prun.sh 16

제 1 절 머리말

1.1 Disclaimer

우리는 이 문서에서 어떤 부정확한 정보에 대한, 혹은 이를 적용합으로서 일어날수 있는 어떠한 피해에 대한 책임을 지지 않는다. We will not accept any responsibility for any incorrect information within this document, nor for any damage it might cause when applied.

1.2 판권

Copyright ? 1997 - 1998 Jacek Radajewski and Douglas Eadline. 이 문서의 배포와 수정의 권한은 GNU General Public Licence를 따른다.

1.3 이 문서에 대해

Jacek Radajewski는 1997년에 이 문서에 대한 작업을 착수하였고, 곧 Douglas Eadline와 같이 동참하였다. 몇 달이 지난후 베오울프 HOWTO는 많은 양의 문서로 늘어 났고, 1998년 8월에 메어울프 HOWTO, 베오울프 구조 디자인 HOWTO, 그리고 베오울프 설치와 관리 HOWTO로 세 종류의 문서로 나누어 졌다. 베오울프 HOWTO 버전 1.0.0은 1998년 11월에 Linux Documentation Project(LDP)로 발표 되었다. 우리는 이 작업이 완벽한 베오울프 문서화 프로젝트로 가는 초석이 되기를 바랄뿐이다.

1.4 저자들에 관해

Jacek Radajewski는 네트워크 관리자로서 일하고 있고, 오스트레리아의 Southern Queensland 대학에서 컴퓨터 과학분야에서 명애 학위(?)를 받고 있다. Jacek의 리눅스와의 첫 만남은 1995년이고, 첫 눈에 반했다고 한다. Jacek는 첫 베오울프 클러스터를 1997년 5월에 만들었고, 그 이후로 계속 이 기술을 사용하고 있다. 항상, 새롭고 더 좋은 설치 방법을 찾고 있다. *jacek@usq.edu.au* `<mailto:jacek@usq.edu.au>`로 e-mail를 보내면 Jacek와 연락을 취할 수 있다. Douglas Eadline는 미국 팬실베니아(PA?) 베드레햄의 Paralogic에서 Principal Scientist이자 President로 Ph.D이다. 그는 Physical/Analytical Chemist로서 해왔었고, 1978년에 화학 실험장치와 사용할 싱글 보드 컴퓨터를 처음 제작하면서 부터 능력을 향상시켜 왔다. Eadline 박사의 관심 분야는 리눅스, 베오울프 클러스터들, 그리고 병렬 알고리즘들을 접목 시키는 것이다. Eadline 박사와 연력을 취하고 싶다면 *deadline@plogic.com* `<mailto:deadline@plogic.com>`으로 email을 보내면 된다.

1.5 감사의 말

베오울프 HOWTO를 저작에 있어서 많은 시간이 걸렸고, 결국 완성을 했다. 이에 도와주신 분들에 감사를 드린다. 이 HOWTO에 다음 사람들의 도움과 기여에 감사 드리고 싶다. Becky 그녀의 사람, 지원과 이해. Tom Sterling, Don Becker, 그리고 NASA에 있는 다름 사람들, 이들 덕에 베오울프 프로젝트가 시작되었다. Thanh Tran-Cong과 Faculty of Engineering and Surverying, topcat 베오울프 머신을 실험해줄 수 있게 해 주었다. 나의 supervisor인 CHristopher Vance, 많은 주된 생각을 주었다. 나의 친구 Russell Waldron의 훌륭한 프로그래밍 착상, 이 프로젝트에 대한 전반적인 관심과 지원을 해주었다. ? My friend Russell Waldron for great programming ideas, his general interest in the project, and support. 나의 친구 David Smith, 이 문서를 교정 해주었다. 베오울프 메일링 리스트에 많은 여러 사람들, 나에게 피드백과 생각을 제공해주었다. 리눅스 운영체제, 그리고 topcat과 다른 베오울프 머신들이 사용하고 있는 공개 소프트웨어 패키지를 담당하는 모든 분들에게 감사를 드린다.

제 2 절 개요

컴퓨터와 네트워크 하드웨어의 성능 향상에 따라서 이들의 가격도 감소하게 되고, 매우 비싼 슈퍼컴퓨터들의 CPU 시간을 사는 것 보다 특별 주문이 아닌 구성 요소들로 부터 병렬 계산에 관한 시스템들의 구성이 더욱더 실용적으로 되어 졌다. 사실, 베오울프 형태의 머신의 성능비율당 가격은 전통적인 슈퍼컴퓨터 보다 3-10배 정도 더 우수하다. 베오울프 구조 규모는 적당하다. 쉽게 구성할 수 있으며, 단지 하드웨어 비용만 지불하고 대부분의 소프트웨어 비용은 무료이다.

2.1 누가 이 HOWTO 문서를 읽어야 하는가?

이 문서는 최소한 리눅스 운영체제에 접해있는 사람을 위해 만들어 졌다. 베이울프 기술의 지식이나 더 복잡한 운영체제의 이해와 네트워킹 개념은 필수적인 것은 아니다. 그렇지만, 병렬계산을 할 수 있는 어느 정도의 상황은 되어 있어야 한다(결국 어떤 이유로 이든지 이 문서를 읽을 테지만). 이 문서는 베오울프에 관한 모든 가능한 질문에 대해 대답해 줄 수는 없지만, 잘만 되면, 착상을 주고 올바른 방향으로 제시해줄 것이다. 이 문서의 목적은 기본 정보, 링크들과 좀더 발전적인 참고 문헌들을 전달하는데 있다.

2.2 베오울프란?

베오울프: far flew the boast of him, son of Scyld, in the Scandian lands. So becomes it a youth to quit him well with his father's friends, by fee and gift, that to aid him, aged, in after days, come warriors willing, should war draw nigh, liegemen loyal: by lauded deeds shall an earl have honor in every clan. (역자주:이부분은 서사시 부분, 번역에 한계가 ^^;) 베오울프는 일찍이 고대 영어로 쒸어진 서사시이다. Grerdel이라는 괴물을 막아낸 매우 힘이 쎄고, 용감한 영웅에 대한 이야기이다. 베오울프 영웅에 대해 더 알고 싶으면 "역

사"부분을 찾아 보아라. 베오울프 슈퍼컴퓨터 시설들을 만들거나 사용하는 사람들 만큼이나 많은 베오울프 정의가 있을 것이다. 몇몇 사람들은 NASA의 고유의 머신과 같은 방법으로 만들어진 그들의 시스템만을 베이울프라고 주장한다. 극단의 다른 사람들은 병렬 코드을 수행하는 어떤 워크스테이션도 베이울프라고 하기도 한다. 나의 베오울프의 정의는 위의 기술한 두가지 관점의 철충으로, 베어 울프의 메일링 리스트에 포스팅된 글을 바탕으로 기술하면:

베오울프는 병렬 계산을 위해 사용될 수있는 다중 컴퓨터 구조이다. 이는 이더넷이나 다른 네트워크를 통해 서로 연결된 하나의 서버 노드와 하나 이상의 클라이언트 노드로 구성되어진 시스템이다. 이는 리눅스를 돌리 수 있는 PC와 같은 일반적인 하드웨어 구성들, 표준 이더넷 어뎁터들, 그리고 스위치들로 이루어진다. 이는 어떤 주문형 하드웨어 구성요소와 trivially reproducible를 포함하지 않는다. 또한, 베오울프는 리눅스 운영체제, Parallel Virtual Machine (PVM)과 Message Passing Interface (MPI)와 같은 일반적인 소프트웨어 를 사용한다. 서버 노드는 모든 클러스터들을 조절하고 클라이언트 노드에게 파일을 제공한다. 더욱이, 클러스터의 콘솔과 외부로 통하는 게이트웨이도 제어를 한다. 거대한 베오울프 머신 하나 이상의 서버 노드를 가지고 가능하다면 다른 노드들이 콘설이나 모니터링 스테이션과 같은 특정 작업들을 맡아서 처리하게 한다. 대부분의 경우 베오울프 시스템에서 클라이언트 노드들은 벙어리이고, 이들은 더 벙어리일수록 더욱 좋다. 노드들은 서버 노드에 의해 조절되고 관리된다. 그리고, 단지 그들이 무엇을 할려고 하는지는 말하기만 하면 된다. 디스크가 없는 클라이언트 설정에서 클라이언트 노드들은 이들의 IP 주소와 이름을 서버가 알리기 전까지 이것들 조차 알필요가 없다. 베오울프와 Cluster of Workstations (COM)과의 가장 큰 차이점은 베오울프는 여러 개의 워크스테이션들 보다 하나의 머신 형태로 작동한다는 사실이다. 많은 경우의 클라이언트 노드들은 키보드나 모니터를 가지고 있지 않고, 단지 원격 로그인이나 가능하다면 직렬 터미널를 통해서만 엑세스할 수 있다. 베오울프 노드들은 마더보드에 CPU나 메모리 모듈을 꽂아넣는 형태처럼 클러스터에 꽂아넣을 수 있는 CPU + 메모리 팩키지 같은 것이라 할 수 있다.

베오울프는 특정한 스프트웨어 팩키지, 새로운 네트워크 위상이나 최신 커널을 파해친것도 아니다. 베오울프는 병렬형태의 가상 슈퍼컴퓨터로 리눅스 컴퓨터들을 클러스터링한 기술이다. 또한, 베오울프를 더 빠르고, 더 쉽게 구성하고, 더 유용하게 할 커널 변경, PVM과 MPI 라이브러리, 그리고 설정 툴들과 같은 많은 소프트웨어 패키지들이 있을 뿐이다. 어떤 추가적인 소프트웨어 없이 이를 이용해서 표준 리눅스 배포을 사용하여 베오울프 클래스 머신을 구성할 수 있다. 당신이 네트워크에 연결된 두 대의 리눅스 커퓨터들을 가지고 있고, 최소한 /home 파일 시스템이 NFS를 통하여 공유하고 있고, 원격 쉘(rsh)를 실행할 수 있을 만큼 신용하고 있고 있다면, 당신은 하나의 간단한 2 노드의 베오울프 머신을 가지고 있다고 말할 수 있다.

2.3 분류

베오울프 시스템들은 다양한 부분들로 구성되어져 왔다. 성능 향상의 목적 을 가진 비-일반적인 구성요소 (하나의 제조업자가 독단적으로 생산한 것) 들도 수용해 왔다. 시스템들의 다른 타입들를 설명하거나 좀 더 쉽게 머신에 대해 토론하기 위해서, 우리는 다음과 같은 간단한 분류안을 제안한다: CLASS I 베오울프: 이 머신의 클래스는 일반적인 "특별 주문이 아닌" 부분으로 구성된다. 일반적인 "특별 주문이 아닌" 부분들을 정의하기 위한 "Computer Shopper" 증명 검사를 사용할 것이다. (Computer Shopper는 PC 시스템들과 구성요소에 대한 1인치 두께의 월간 잡지/목록을 말한다.) 이 시험은 다음과 같다:

```
CLASS I 베오울프는 최소한 세 개의 국가적으로/전세계적으로 배부된 광고
  목록들에서 발결할 수 있는 부품들로 부터 조립 가능한 머신을 말한다.
CLASS I 시스템의 이점:
   여러가지 제품들을 이용하여 구성할 수 있는 하드웨어 ( 낮은 가격, 쉬운
     유지)
   하나의 하드웨어 회사에 의존하지 않음
   리눅스 commodity에서 지원되는 드라이버
   일반적으로 표준안을 기본을 둔 것 (SCSI, 이더넷, 등등.)
CLASS I 시스템의 단점:
   최고의 성능을 위해서는 CLASS II 하드웨어를 요구
CLASS II 베오울프
```

```
CLASS II 베오울프트는 다만  Computer Shopper 인증 시험에 통과하지 않은
  어떤 머신이라도 된다. 이는 나쁜 것만은 아니다. 실제로는, 단지 머신의
  분류일뿐 이다.
CLASS II 시스템의 이점:
   매우 좋은 성능을 보여줌!
CLASS II 시스템의 단점:
   드라이버 지원이 변동적
   하나의 하드웨어 회사에 의존적
   CLASS I 시스템들보다 그가
```

하나의 CLASS는 다른 것 보다 더좋을 필요는 없다. 이는 당신의 필요와 예산 에 따를뿐이다. 이런 분류 시스템은 베오울프 시스템들에 관한 논의를 더욱 간결하게 하기위한 것이다. "시스템 디자인"부분에서 당신의 요구에 가장 적합한 시스템의 종류가 무엇인지 결정하는데 도움을 줄 것이다.

제 3 절 구조 개요

3.1 무엇 처럼 보이는가?

내 생각에는 베오울프 슈퍼컴퓨터 구조를 기술하기위한 가장 좋은 방법은 실제 베오울프와 매우 비슷한 예를 드는 것이다. 그러나, 대부분의 시스템 관리자들에게 익숙한 것이어야 한다. 그 예로, 베오울프 시스템과 가장 근접한 것이 하나의 서버와 여러개의 클라이언트들로 구성된 유닉스 컴퓨터 연구실 이다. 좀 더 구체적으로 말하면, 예를 들어 나는 USQ의 Faculty of Sciences에 컴퓨터 연구실에 재학중에 DEC Alpah을 사용하려한다고 하자. 이 서버 컴퓨터는 beldin이라 하고 클라이언트 머신들은 scilab01, scilab02, scilab03에서 scilab20까지 있다고 하자. 모든 클라이언트들은 Digital Unix 4.0 운영체제의 지역 카피본으로 설치되어있지만, 사용자 파일 공간 (/home)과 /usr/local은 NFS (Network File System)를 통해서 공유되어 진다. 각각의 클라이언트 각 /etc/hosts.equiv 파일에 서버나 다른 모든 클라이언트들 에대한 엔트리가 포함되어 있어서, 모든 클라이언트는 다른 곳으로 원격 쉘 (rsh)를 수행할 수 있다. 서버 머신은 모든 연구실에 대한 NIS 서버로서, 다른 머신들을 끼리 계정 정보를 공유한다. 어떤 사람이 scilab02 콘솔에 앉아서, 로그인하면, 그가 서버나 scilab15에 접속한다 해도 같은 환경을 가지게 된다. 이러한 이유로 모든 클라이언트는 똑같은 관점과 느낌을 가진다. 이 이유는 모든 머신에 같은 방법으로 운영체제가 설치되었고 설정 되었있고, 사용자의 /home과 /usr/local 영역 둘다 물리적으로는 서버에 있지만 클라이언트는 NFS를 거쳐서 엑세스 할 수 있기 때문이다. NIS나 NFS에 대한 더 많은 정보를 원하면 NIS와 NFS HOWTO를 읽어보기 바란다.

3.2 다른 노드를 어떻게 이용할 것인가?

지금부터 우리는 시스템 구조에 대한 어떤 생각을 가지고 있고, 컴퓨터 연구실에 있는 머신의 사용가능한 CPU 사이클들을 사용할 수 있는 방법을 찾아보기로 하자. 어떤 사람이 임의의 머신에 로그인 할 수 있고, 그들의 홈 디렉토리에서 프로그램을 실행할 수도 있지만, 같은 일을 원격 쉘을 가지고 여러 개를 실행 할수도 있다. 예를 들어, 우리가 1에서 10을 포함한 모든 정수 값의 평방근의 합을 계산할려고 한다. 이를 완벽히 실행할 sigmasqrt("소스 코드" 부분을 보아라)라는 간단한 프로그램을 작성을 한다. 1에서 10까지의 평방근 의 합을 계산하는 것을 실행하면:

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 10
  22.468278
real    0m0.029s
  user    0m0.001s
  sys     0m0.024s
```

time 명령어는 이 작업 실행하는 wall-clock(끝마치는 시간)을 확인할 수 있다. 위에 보이듯이, 이 예제는 실행하데 작은 비율의 시간(0.029초)만 걸린다. 그러나, 1에서 1 000 000 000까지의 정수값의 평방근 합을 구

할려고 하다면 어떻게 될까? 이것을 가지고 wall-clock 시간을 다시 구해보자.

```
[jacek@beldin sigmasqrt"]$ time ./sigmasqrt 1 1000000000
  21081851083600.559000
real    16m45.937s
  user     16m43.527s
  sys      0m0.108s
```

이 프로그램의 실행 시간은 꽤 길어졌다. 그러면, 어떻게 하면 일을 수행 하는 시간을 좀 더 빨리 할 수 없는가하는 질문을 하게될 것이다. 일을 어떻게 바꾸어야 wall-clock 시간이 최소로 될까? 확실한 대답은 일을 여러 개의 하위 일들로 나누어서 각각의 하위 일들을 모든 컴퓨터들에서 병렬로 수행하는 것이다. 우리는 작업을 크게 20개 부분으로 나누어서, 평방근의 한 범위를 계산해서 각각의 노드에 이 값을 더하게 했다. 모든 노드가 작업을 마치고 값을 반환할때, 20개의 결과를 모두 더해서 마지막 해를 얻게된다. 일을 수행하기 전에 모든 프로세스가 값들을 기록할 명명 파이프(named pipe)를 생성켰다.

```
[jacek@beldin sigmasqrt"]$ mkfifo output
  [jacek@beldin sigmasqrt"]$ ./prun.sh & time cat output | ./sum
  [1] 5085
  21081851083600.941000
  [1]+  Done                    ./prun.sh
real    0m58.539s
  user     0m0.061s
  sys      0m0.206s
```

약 58.5 초의 시간을 얻었다. 이 시간은 일이 시작할때 부터 모든 노드들가 계산을 마치고 각 결과를 파이프에 기록하는 시간까지를 말한다. 이 시간은 마지막 20개의 덧셈은 포함하지 않았지만, 매우 작은 시간이므로 무시할 수 있다. 우리는 병렬로 일을 수행함에서 의미심장한 향상이 있어음을 볼 수 있을 것이다. 위의 예제의 목적은 가장 간단한 병렬화되는 병행 코드의 방법을 보여주자는데 있다. 실제로 이와 같은 예는 매우 드물고 다른 기술 (PVM과 PMI API들)이 병행성을 이루는데 사용되지곤 한다.

3.3 베오울프가 COW와 어떻게 다른가?

앞에서 기술한 컴퓨터 연구실은 Cluster of Workstations(COW)의 완벽한 예이다. 그래서, 베오울프의 특징이 무엇이고 COW와 어떻게 다른가?를 알아 보자. 그렇게 다를 것은 없지만, 베오울프트는 약간의 독특한 특징들을 가진다. 먼저, 베오울프 클러스터에서 모든 클라이언트 노드들의 경우에는 키보드, 마우스, 비디오 카드, 그리고 모니터 조차 없다. 서버 노드, 콘솔 전용 노드, 또는 시리얼 콘솔에서의 클라이언트 노들의 모든 엑세스는 원격 접속을 통해서 이루어진다. 클라이언트 노드가 클러스터 외부의 머신을 엑세스하거나, 클러스터 외부의 머신이 클라이언트 노드를 직접 엑세스할 필요가 없기 때문에, 클라이언트 노드는 10.0.0.0/8 이나 192.168.0.0/13 주소 범위 ((RFC 1918 *www.alternic.net/rfcs/1900/rfc1918.txt.html* `<http://www.alternic.net/rfcs/1900/rfc1918.txt.html>`)와 같은 사설 IP 주소들이 일반적으로 많이 사용한다. 보통 두 번째 네트워크 카드로 외부로 연결되는 머신은 서버 노드만이 할 수 있다. 시스템을 사용하는 가장 일반적인 방법은 서버 콘솔에서 직접 엑세스하거나, 개인 워크스템이션에서 텔넷이나 원격 로그인으로 서버 노드를 엑세스한다. 서버 노드 상에서 사용자가 코드를 편집하고 컴파일 하고, 클러스터에 있는 모든 노드들에게 일을 시키게 된다. 대부분의 경우 COW은 밤에 병렬 계산하는데 사용되곤하고, 사람들이 사용하지않은 주말에 사용되기도 한다. 그러므로, 사용하지 않은 CPU 시간을 이용하는 것이다. 다른 관점에서 베오울프는 일반적으로 병렬 계산을 하기 위한 전용 머신이고 이러한 목적에 맞게 최적화되었다. 또한, 베오울프는 off-the-shelf 요소들로 이루어지고 주로 공개 소프트웨어를 사용하기 대문에 가격/성능 비가 매우 좋다. 여러 시스템을 가지고 있는 베오울프는 사용자가 베오울프 클러스터을 바라볼때 하나의 계산 워크스테이션 으로 보이는 특징을 가지고 있다.

제 4 절 시스템 디자인

하드웨어를 구입하기 전에 시스템의 디자인을 고려해보는 것이 좋다. 베오울프 시스템의 디자인을 하기 위한 기본적인 2가지 하드웨어 문제점로는: 당신이 사용할 노드나 컴퓨터의 종류; 컴퓨터 노드들을 연결하기 위한 방법. 하드웨어를 결정하는데 있어서 영향을 미치는 소프트웨어 문제점이 통신 라이브러리나 API 이다. 소프트웨어에 대한 자세한 사항은 이 문서 후반부에서 다루기로 하겠다. 선택의 폭이 그렇게 넓지는 않은 반면에, 베오울프 시스템을 구성할때 몇 개의 중요한 디자인 결정이 있다. 병렬 계산의 과학(또는 예술)은 많은 다른 해석이 있을 수 있기때문에, 개요는 다음에 제공하겠다.(?) 배경이 되는 요소를 읽고 싶지 않으면, 이 절을 뛰어 넘어도 좋지만, 마지막 하드웨어 결정을 하기 전에 "적합성"이라는 절을 읽는 것을 권하는 바이다.

4.1 병렬 계산에있어서 배경 요약.

이 절은 병렬 계산 개념에 대한 배경지식을 제공할 것이다. 병렬 계산 과학과 기술을 완벽히 다루지는 않을 것이다. 베오울프 디자이너와 사용자에게 중요한 문제점에 대한 간략한 기술을 할 것이다. 당신의 베오울프를 디자인하고 구성함에 있어서, 당신의 결정과정에 있어서 다음에 기술될 많은 문제점이 중요 하게 되게 될 것이다. 베오울프 구성요소의 특성으로 인해서 베오울프 슈퍼컴퓨터는 많은 인자들이 사용자의 제어하에 있기 때문에 조심스럽게 고려하는 것을 요구한다. 일반적으로, 병렬 계산과 관련된 문제점들을 이해하기 어려운 것만은 아니다. 사실, 문제점을 이해하기만 하면, 당신의 기대는 더욱 현실적이게 되고 성공은 더욱 현실과 가까워지게 될 것이다. 가장 중요한 인자인 프로세서 속도는 "순차 세계"와는 달라서, "병렬 세계"에서 프로세서 속도는 전체 시스템 성능과 효율를 결정하는 몇가지 인자중에 하나이다.

4.2 병렬 계산 방법

병렬 계산은 많은 형태를 가지고 있다. 사용자 관점에서 각각의 방법론의 장점과 단점를 고려해보는 것이 중요하다. 다음 절은 병렬 계산의 방법상의 몇가지 관점을 제공하고, 머신이 일련의 과정중에 어느 곳의 베오울프 머신이 실패하였는지를 가르치려고 한다.

4.2.1 왜 한 개 이상의 CPU가 필요한가?

이 질문의 대답은 중요하다. 8개의 CPU를 가지고 워드프로세서를 실행한다고 하면 "over-kill"처럼 들리고, 또한 그렇다. 웹 서버, 데이터베이스, 랜더링 프로그램, 또는 프로젝트 일정표에 대해서는 어떠한가? 여분의 CPU가 있다면 도움이 될 것이다. 복잡한 시뮬레이션, 유체역학 코드, 또는 데이타 마이닝 풀그림은 어떠한가? 이런 상황에서는 여분의 CPU는 찐짜루 도움이 된다. 사실, 다중 CPU은 더욱더 많은 문제들을 풀기위해 사용되지곤 한다: 보통 다음 질문 으로: "왜 두 개 또는 네 개의 CPU가 필요한가, 나는 986 터보-하이퍼 칩이 나오기를 기다리려고 한다." 이러한 이유로는:

1. 멀티 테스킹 운영 체제의 사용으로, 몇가지의 것들을 한 번에 실행 할 수 있다. 하나 이상의 저렴한 CPU를 가지고 쉽게 사용할 수 있는 고유의 "병렬성"을 말한다.
2. 프로세서 속도는 매 18 개월 마다 두 배로 향상되어 왔다. 그러나, 메모리 속도나 하드 디스크 속도는 어떠한가? 불행하게도 이런 속도는 CPU 속도 만큼이나 빠르게 증가히지 못했다. 주목 할 것은 모든 풀그림은 "out of cache memory access"와 하드 디스크 엑세스를 요구한다는 것이다. 병행에 있어서 이런 것을 하기 위해서는 몇가지 제한점을 회피하는 방법을 사용 해야 한다.
3. 예상하기를 프로세스 속도는 2005년 이후로는 18개월 마다 2배로 증가 하지 못할 것이라고 추측한다. 이런 경향을 극복하기위해서는 매우 심각한 장애물들이 놓여있다.
4. 풀그림에 따라서, 병렬 계산은 2에서 500배정도까지 더 빠르게 속도를 향상 시킬 수 있다(몇몇의 경우 더 빠르다). 이와 같은 성능은 단일 프로세서를 사용하는 곳에서는 불가능하다. 슈퍼컴퓨터 조차

도 매우 빠른 주문형 프로세스을 사용하는 것에서 다중 "commodity-off-the-shelf" CPU를 가지고 만들고 있다.

계산 한계 문제점 때문이나 I/O 한계 문제점으로 인해서 속도를 요한다면, 병렬을 생각해보는 것이 바람직하다. 병렬 계산이 다당한 방법으로 구현이 가능하기 때문에 병렬로 문제를 해결하는 것은 매우 중요한 결심을 해야할 것이다. 이러한 결정은 휴대석, 성능, 그리고 풀그림의 가격에 대한 효과를 극대화할 수 있다. 기술적으로 들어가기 전에, 친숙한 예제인 상점에서 길게 늘어서 줄에서 대기를 가지고 실제 "병렬 계산 문제점"에 대해 다루어 보겠다.

4.2.2 병렬 처리 가계

아주 큰 가계 앞에 8개의 현금 계산대로 함께 있다고 하자. 각각의 계산대는 하나의 CPU이고 각 고객은 컴퓨터 프로그램이라 가정하자. 컴퓨터 프로그램의 크기(일의 양)은 각 고객의 요구의 크기이다. 다음 analogy들은 병렬 계산 계념을 설명할때 사용되곤 한다.

Single-tasking Operating System 하나의 현금 계산대는 열려이고(사용중) 한번에 하나의 고객을 처리한다.

예: `MS DOS`

Multi-tasking Operating System: 하나의 현금 계산대는 열려 있지만, 현재 한번에 각 요구의 부분만을 처리 한고 다음 사람의 요구의 일부를 처리한다. 모두가 줄을 같이 지나가는 것 처럼 보일테지만, 아무도 줄에 없다면, 더 빨리 지나가게 될 것이다.

예: `UNIX, NT using a single CPU`

Multitasking Operating Systems with Multiple CPUs: 현재 가게에 몇개의 현급 계산대가 열려있다. 각 요구는 따로따로 현금 계산대가 처리하고 줄은 더 빨리 이동하게 된다. 이를 SMP (Symetric Multi-processing)라 한다. 여분의 현금 계산대다 열려 있을 지라도, 한개의 현금 계산대를 사용하는 것 보다 더 빨리 지나가지는 못한다.

예: `UNIX and NT with multiple CPUs`

Threads on a Multitasking Operating Systems extra CPUs 요구 중에 "break-up"이라는 아이템이 있다면, 한번에 여러 현금 계산대를 사용하여 줄을 더 빠르게 이동할 수 있다. 먼저, 여러개의 현금 계산대를 사용할려면 "당신의 요구를 조각냄"시간을 들여야 하기 때문에 많은 양의 물건을 가지고 있다고 하자. 이론적으로는 전보다 "n"배 정도 더 빠르게 줄을 통과할 수 있다(여기서 "n"은 현금 계산대 개수). 출납원이 총 합계를 구 할때, 다른 "지역" 현금 계산대를 보거나 말을 하여 재빠르게 정보를 교환 한다. 그들이 정보를 찾기 위해 다른 현금 계산대를 기웃거릴 지라도, 그들은 더 빨리 일을 처리하기를 원한다.(?) 그러나 얼마나 많은 현급 계산대가 가게에 있어야 하고 어느 장소에 위치해야 효과적인가하는 제한에 부딪치게 된다. Amdals 법칙에서 풀그림의 속도향상은 프로그램의 가장 늦은 일련의 부분으로 제한해야 된다는 것이다. 예: 같은 마더보드에서 여분의 CPU를 가진 UNIX 나 NT에서 다중 쓰레드화된 프로그래들 수행.

Sending Messages on Multitasking Operating Systems with extra CPUs: 성능 향상에 있어서, 가계 뒤에 8개의 계산대를 추가하였다. 새로운 현급 계산대는 정면 현급 계산대와는 떨어져 있기 때문에, 출납원은 이들의 총 합계을 상점 정면으로 보내기 위해서는 전화를 사용하여야 한다. 이런 거리는 출납원 사이의 통신에 있어서 추가적인 오버헤드(시간)를 들게 한지만, 통신을 적게한다면, 이것은 문제도 아니다. 모든 계산대를 필요로 할 만큼 진짜루 큰 요구가 있다면, 같은 시간에 모든 현금 계산대를 사용하여 속도를 향상시키기 전에 추가적인 오버헤드가 발생할 수 있음을 고려해야 한다. 몇몇의 경우에 모든 상점을 통

틀어 한개의 현금 계산대(또는 현금 계산대 섬)를 가지고 있다고 볼수 있다(각각의 현금 계산대(또는 섬)은 전화를 통해서 통신해야 한다). 현금 계산대에 작업하는 모든 출납원은 전화를 통해서 각자 얘기를 할 수 있다면, 그들이 어디에 있든지 상관이 없다.

```
예: One or several copies of UNIX or NT with extra CPUs
  on the same or different motherboard communicating through messages.
```

위의 시나리오가 정확하지 않을 지라도 병렬 시스템의 제한점을 잘 표현 하고 있다. 한개의 CPU(또는 현금 계산대) 통신도 다른 문제점이다.

4.3 병렬 계산을 위한 구조

병렬 계산의 보편적 방법과 구조은 다음에 나타나 있다. 완전하게 설명하지 않을 지라도, 베오울프 디자인에 관련된 기본적인 문제점을 이해하는데 충분하다.

4.3.1 하드웨어 구조

다음 두 방법을 사용하는 기본적인 병렬 컴퓨터:

1. 메시지를 가지고 통신하는 지역 메모리 머신(베오울프 클러스터들)

2. 메모리를 통해 통신하는 공유 메모리 머신들(SMP 머신들) 전형적인 베오울프는 고속 이더넷을 사용하여 단일 CPU 머신을 연결한 집합이기에 지역 메모리 머신이기도 하다. 4 way SMP 박스는 공유 메모리 머신이고 병렬 계산으로 사용되지고 있다. - 병렬 풀그림은 공유 메모리를 이용하여 통신한다. 컴퓨터 가계에서 유추한다면, 지역 메모리 머신들 (개개의 현금 계산대)은 많은 수의 CPU들로 확장할수 있는 반면에 공유 메모리 머신들의 CPU수는 메모리 연결로 인하여 제한적이다. 그러나, "혼합" 공유 메모리 머신으로 만들면 많은 수의 공유 메모리 머신들을 연결할 수 있다. 이런 혼합 머신은 사용자에게는 한개의 거대한 SMP 머신으로 보이고, 글로발 메모리는 프로그래머에 의해 보여지고 모든 CPU들에의해 공유되어 다른 레턴시들을 가지고 있기 때문에 NUMA (non uniform memory access) 머신이라도 한다. 그렇지만, 몇개의 단계에서 NUSA 머신은 지역 공유메모리 풀들 사이에서 메시지를 통과하여야만 한다. 또한, 노드들을 계산하는 지역 기억장소로서의 SMP 머신으로 연결이 가능하다. 전형적인 CLASS I 마더보드는 2 개나 4 개의 CPU를 가지고 전반적인 시스템 비용을 감소하는데 사용된다. 리눅스 내부 스케줄러는 어떻게 CPU들을 공유할지를 결정한다. 사용자는 (이 시점에서) 특정 SMP 프로세서에서 특정 일을 할당할 수 없다. 그러나, 사용자는 두개의 독립된 프로세스나 쓰레드된 프로세스들 실행할 수 있고, 한개의 CPU 시스템에서 성능 향상을 기대할 수 있다.

4.3.2 소프트웨어 API 구조

프로그램에서 병행성을 표현하는 기본적인 표현 두 가지 방법:

1. 프로세서들간 메시지 전송 사용

2. 운영체제 쓰레드 사용 다른 방법도 있지만, 위의 2 가지가 폭넓게 사용된다. 병형성의 표현은 하드웨어의 기초로 조절할 필요는 없다는 것을 기억하는 것이 중요하다. 메시지나 쓰레스 둘다는 SMP, NUMA-SMP, 그리고 클러스터에 의해 구현되어져 있다. - 나중에 효율적이고 휴대성에대해 설명하겠고 이는 중요한 논쟁점 이다.(?)

메시지 역사적으로, 메시지 전송 기술은 초기 지역 메모리 병렬 컴퓨터 디자인에서 가져왔다. 쓰레드는 같은 장소에서 데이터를 사용하는 반면에 메시지는 데이터 복사를 가져다 쓴다. 복사되어 질수 있는 메시지들에서 대기시간와 속도은 메시지 전송 모델에 따라 제한 인자가 있다. 메시지는 아주 간단하다. : 몇몇

데이터와 목적 프로세서. 공통 메시지 전송 API들은 PVM또는 MPI이다. 메시지 전송은 쓰레드와 메시지 가지고 효과적으로 구현된다. 둘다 SMP 머신과 머신들의 클러스터 사이에서도 잘 작동한다. SMP 머신에서 메시지를 사용함 으로서 얻어지는 이점은 쓰레드 반대로 앞으로 클러스터들을 사용할 결심을 했다면 머신들의 추가나 풀그림 확장을 쉽게 할 수 있다.

쓰레드 운영 체제 쓰레드는 공유 메모리 SMP (Symmetrical Multiprocessing)은 프로그램 병행 부분들 사이에서 더 빠른 공유 메모리 통신과 동기화 을 가능하게 디자인 하기 위하여 개발되겠다. 쓰레드는 공유 메모리를 통하여 통신을 하기 때문에 SMP 시스템들에서 잘 작동한다. 이러한 이유로 사용자는 글로발 데이터에서 지역 데이터를 분리해야 한다. 그렇지 않으면, 프로그램은 잘 작동하지 않는다. 메시지에 비교하면, 프로세스들(쓰레드들) 상에 데이터를 공유하기 때문에 많은 양의 복사가 생략된다. 그렇지만, cache coherence 이라는 오버헤드가 덧붙이게 되었다. 쓰레드는 SMP 경계 저편으로 연장되는 NUMA 기술을 요한다. 이 기술은 사치스럽고 전에 리눅스에 의해 지원되지 안았었다. 메시지 위에서 쓰레드를 구현해 오고 있지만 (*syntron.com/ptools/ptools_pg.htm* `<http://syntron.com/ptools/ptools_pg.htm>`), 쓰레드가 메시지를 이용하여 구현하였을 때 종종 비효율적이게 된다. 다음은 성능에 관한 상태를 보여주고 있다:

	SMP 머신 성능	머신들의 클러스터 성능	확장성
메시지	좋음	매우 좋음	매우 좋음
쓰레드	매우 좋음	나쁨*	나쁨*

* 사치스런 NUMA 기술을 요구.

4.3.3 풀그림 구조

다중 CPU에서 병렬로 풀그림을 실행하기 위해서는 병행 부분으로 확실히 쪼개야 한다. 표준 단일 CPU 풀그림은 다중 프로세서에서 단일 CPU 풀그림 보다 더 빠르지는 않을 것이다. 프로그램을 쪼개져야할 몇가지 툴들과 컴파일하지만, 병렬 코드은 "plug and play" 작동을 하지 않는다. 풀그림에 의존적으로 병렬 코드는 쉽고, 극도록 어렵거나, 알고리즘 의존에 따라 불가능한 경우도 있을 수 있다. 소프트웨어 문제점의 설명이 필요한 적합성 개념을 짚고가자.

4.4 적합성

병렬 계산에 대한 대부분의 질문들은 비슷한 답변을 가진다: "모든 것은 풀그림에 달렸있다." 논쟁점으로 들어가기 전에, 만들어질 필요가 있는가에 대한 매우 중요한 구별이 하나 있다 - 병행과 병렬사이의 차이. 이런 논의를 한 목적은 다음에 오는 두 가지 개념을 정의하기 위해서 이다: 프로그램의 병행 부분은 독립적으로 계산이 되는 것이다. 프로그램의 병렬 부분들은 그들의 병행 부분으로 같은 시간에 분리된 처리 단위에서 실행된다. 이런 구별은 매우 중요하다. 왜냐하면, 병행성은 프로그램의 속성이고 효율적인 PARALLELISH 은 머신의 속성이기 때문이다. 이상적으로, 병렬 실행은 더 빠른 성능으로 끝마쳐야 한다. 병렬 성능에서 재한 요인은 통신 속도와 계산 노드들 사이의 대기시간이다. (또한, 대기시간은 cache coherency로 인해 쓰레드화된 SMP 풀그림에도 있다.) 많은 일반 병렬 평가 테스트은 높은 병렬, 통신, 그리고 대기시간이지, 병목현상에 있지 않다. 이런 형태의 문제를 "obviously parallel"이라 한다. 다른 풀그림들은 병렬에서 프로그램의 병행 부분이 그렇게 간단하고 실행할 수 있는 것이 아니기에 사실 프로그램이 더 느리게 실행하는 원인이 되기도 한다. 그렇기 때문에 프로그램의 다른 병행 부분에 얻어진 임의 성능향상을 떨어뜨리게 된다. 짧은 기간에서 통신 시간 비용을 계산 시간으로 보충해야 한다. 그렇지 않으면, 병행 부분의 병렬 실행은 비효율적이다. 프로그래머의 임무는 프로그램의 병행 부분들을 병렬로 실행해야 하는 것과 하지 말아야 할 것을 결정하는 것이다. 이 대답은 풀그림의 효율로 결정할 수 있다. 다음 그래프는 프로그램의 상황을 간략히 보여주고 있다.

```
         | *
         | *
         | *
  % of   | *
  appli- |  *
  cations |  *
         |  *
         |  *
         |    *
         |     *
         |      *
         |         ****
         |             ****
         |                 *********************
         +------------------------------------
          communication time/processing time
```

완벽한 병렬 컴퓨터에서 통신/처리 비율이 같이지게 될 것이고, 어떤 것이든지 병렬로 병행을 구현되어 진다. 공유 메모리를 가지고 있는 실제 병렬 컴퓨터에서는 이 그래프에서 보여주는 효과에 따라오게 된다. 베오울프 디자인 할 때, 병렬 효율은 특정 병렬 컴퓨터에 대한 통신 시간과 처리 시간의 비율에 의존적이기 때문에 사용자 이 그래프를 마음에 새겨두기 바란다. 풀그림은 병렬 컴퓨터들 사이에 이동할 수 있어야 하지만, 다른 플랫폼상의 효율은 보장할 수 없다. 일반적으로 이식가능하면서 효율적인 병렬 프로그램은 존재하지 않는다.

위의 그래프를 통해 또 다른 한가지를 유추할 수 있다. 병렬 수행 효율은 통신/처리 비율에 의존하는 것이기 때문에, 단지 이 비율 요소의 한 부분만을 (즉, 통신 성능 또는 처리 성능 둘 중 하나만을) 변화시키는 것으로부터 어떤 특정 응용프로그램은 더 빨라질 것이라는 기대를 항상 할 수 있는 것은 아니다라는 것이다. 통신속도는 변하지 않은 상태에서 처리 속도만의 변화는 프로그램에 예상치 않은 영향을 줄 수 있다. 예를들어, 통신속도는 그대로 유지한채 CPU 속도를 두배 또는 세배 증가시킨 경우 변경 이전 병렬 수행이 효율적이었던 프로그램의 부분을 순차적으로 수행하도록 수정하는 것이 더욱 효율적인 경우도 있다. 말하자면 이전의 병렬실행부분을 이제는 순차적으로 수행시키는 것이 속도가 더 빠를 수 있다는 것이다. 더 나아가, 병렬화하면 비효율적인 부분을 병렬수행하게 되면 결국 응용프로그램이 최고 속도를 내지 못하게 된다. 이러한 경우, 더 빠른 프로세서를 추가하여 병렬수행시키면 오히려 응용프로그램의 속도가 떨어지게 된다.(그 응용프로그램이 최고 속도를 내도록 하기 위해서는 새로운 CPU를 추가하지 말아야 한다.) 그래서, 결론으로 병렬 하드웨어 환경을 사용해야 할지 하지 말아야 할지를 알고 있어야 한다. 그렇기 때문에, 병렬 머신의 적합성이 풀그림에 맞는지 간파해야할 필요가 있다. CPU 속도, 컴파일러, 메시지 전송 API, 네트워크 등등을 포함한 많은 논쟁점을 살펴볼 필요가 있다. 풀그림의 윤곽일 뿐이지 전체적인 이야기를 주는 것이 아님을 주의하라. 프로그램의 중요한 부분을 구분할 수 있지만, 이 부분에 대한 통신 비용을 알 수 는 없다. 주어진 시스템대해서도, 통신 비용은 코드 효율을 병렬화할 수는 없다. 마지막으로 일반적인 오해를 보도록 하자. 자주 "병렬화된 프로그램"이라 하지만, 실제로는 프로그램의 CONCURRENT 부분으로만 위치하고 있다. 효과적인 PARALLELIZATION은 머신의 속성이다.

4.5 병렬 소프트웨어 작성과 포팅

병렬 계산이 필요한다고 결정했고 베오울프를 디자인하고 구성하기를 원한다면, 잠시 전의 논의된 내용을 살펴 보는 것이 좋을 것 같다. 일반적으로 다음 두 가지를 해야 한다:

1. CLASS I 베오울프를 구성해보고 난후에 당신의 풀그림을 그것에 맞게 작성해 보자. 아니면 베오울프에서 작동하는 병렬 풀그림을 실행보라 (그러나 앞에서 언급했던 이동성과 효율의 논쟁점을 주의해라)

2. 베오울프에 실행할 풀그림을 살펴보고 필요로 하는 하드웨어와 소프트웨어 형태에 대한 평가를 해보아라.

이런 경우 일지라도, 몇몇 관점에서 효과에 대한 문제을 살펴볼 필요가 있다. 일반적으로 다음의 세 가지를 생각 해볼 필요가 있다:

1. 프로그램에서 병행 부분을 결정
2. 병렬 효율을 계산
3. 프로그램에서 병행 부분을 기술 자! 지금부터 이들을 살펴보자.

4.5.1 프로그램의 병행 부분 결정

이 단계는 빈번하게 "프로그램 병렬화하기"이라 하기도 한다. Parallelization 결정은 두 번째 단계에서 할 것이다. 이 단계에서서는 데이터 의존을 결정할 필요가 있다. >실질적인 입장에서, 풀그림은 두 가지 병행 형태를 보이고 있다: 계산 (number crunching)과 I/O (데이터베이스). 많은 경우에 계산과 I/O 병행성은 orthogonal이지만, 풀그림은 둘 다를 요구한다. 기존의 풀그림에 대한 병행 분석을 수행할 수 있는 도구들이 있다. 이들 대부분의 도구들은 포트란에 의해 작성되었다. 포트란을 사용하는 주된 두 가지 이유: 대부분의 crunching 풀그림들이 포트란으로 작성되었고 분석하기가 쉽다. 도구들이 없다면, 이 단계는 기존 풀그림에 대해 다소 어려울 것이다.

4.5.2 병렬 효율 계산

도구들의 도움이 없으면, 시행착오를 거치거나 옛날 지식에 의해 추측을 가지고 해야할 것이다. 특정 풀그림을 생각하고 있으면 CPU 한계(계산 범위)와 하드 디스크 한계(I/O 범위)를 결정해보아라. 베오울프의 요구 사항은 당신의 요구에 따라 매우 다양하다. 예를 들어, 계산 한계에서의 문제은 몇개의 매우 빠른 CPU와 고속의 낮은 대기 시간의 네트워크를 필요로 한다. 반면에 I/O 한계 문제는 더 느린 CPU와 빠른 이더넷이 더 효과적으로 작동을 한다. 이런 권고는 종종 대부분의 사람들을 놀라게 한다. 왜냐하면, 보편적인 생각으로는 더 빠른 프로세서가 항상 더 좋다라고 여기기 때문이다. 예산이 한계가 없다면 맞는 말일지 모르지만, 실제 시스템은 비용에 있어서 제한이 있다. I/O 한계 문제에 대해서 조금 알려진 규칙(Eadline-Dekov 법칙이라 불림)이 있는데 꽤 도움이 된다: 똑 같은 누적 CPU 성능 색인을 가진 주어진 두 개의 병렬 컴퓨터에서, 늦은 프로세서(그리고 아마 프로세스간 통신 네트워크도 느려질것이다)은 I/O가 많은 풀그림에 대해서 더 좋은 성능을 보여줄 것이다. 이런 법칙의 증명은 이 문서의 나중에 나올 것이지만, 흥미가 있다면 Performance Considerations for I/O-Dominant Applications on Parallel Computers (Postscript format 109K) (*www.plogic.com/pub/papers/exs-pap6.ps* `<ftp://www.plogic.com/pub/papers/exs-pap6.ps>`) 논문을 받아서 보면 된다. 당신의 풀그림에서 병행의 형태를 무엇인지를 결정을 했었다면, 병렬로 했을 경우 얼마나 효율이 있는지 계산할 필요가 있다. 소프트웨어 도구에 대한 설명은 "소프트웨어"절을 보아라. 도구를 가지고 있지 않으면 이 단계를 거치는 방법을 추축해야할 것이다. 계한 한계 루프를 분단위로 측정하고 초단위로 데이터 전송을 측정했다면, parallelization를 위한 좋은 후원자가 되어야 한다.(?) 그러나 기억해야 할 것은 16분이 걸리고 32 부분으로 분리하였다면, 그리고 각 부분 마다 데이터 전송하는데 몇 초의 시간이 걸린다면, 그 것은 여유를 가질 수가 없게 될 것이다. 감소하기 위한 반환점으로 도달할 것이다.

4.5.3 프로그램에서 병행 부분을 기술

프로그램의 병병 부분을 기술하기 위한 방법이 몇가지가 있다:

1. 명시적 병행 실행
2. 암시적 병행 실행

이들 간의 주요한 차이점은 명시적 병렬성은 사용자에 의해 결정이 되고, 암시적 병렬성은 컴파일러에 의해 결정이 된다.

명시적 방법 기본적인 방법으로 병렬 컴퓨터를 위해 밑그림의 특정 부분을 사용자가 변경해야만 한다. 사용자는 반드시 PVM이나 MPI을 추가하거나 POSIX 쓰레드를 사용하는 쓰레드를 추가해야한다. (그러나 마음에 기억해두어야 할 것은 쓰레드 SMP 마더보드들 사이로 이동할 수 없다.) 명시적 방법은 구현과 벌레잡기가 매우 어려운 경향이 있다. 전형적으로 사용자가 표준 포트란 77이나 C/C++ 밑그림에서 명시적 함수라 불리는 것을 포함해야 한다. MPI 라이브러리은 몇가지 표준 병렬 방법을 쉽게 구현할 수 있는 몇가지 함수를 가지고 있다.(예로, 흩어주는/모으는 함수들). 추가적으로 병렬 컴퓨터을 위해 쒸여진 표준 라이브러리를 이용하는 것도 가능하다. 그렇지만, 주목할 것은 이동성과 효율성은 모순적이다. 역사적인 이유로해서 많은 숫자의 cruching 코드들은 FORTRAN으로 작성되어 져 있다. 이러한 이유로해서 FORTRAN은 병렬 계산에 있어서 가장 많은 양의 지원(도구들, 라이브러리들, 등등)을 가지고 있다. 현재 C를 사용하고 있는 많은 프로그래머들이 C가 더빨리 실행돼다는 생각으로 기존 FORTRAN 풀그림을 C로 재작성하고 있다. 이러한 생각이 보편적인 머신 코드에서 C로 된 것은 사실일지 몰라도, 몇가지 주요한 결점을 가지고 있다. C에서 포이터의 사용은 데이터 의존성을 결정하는데 있어서 극도록 어렵게 만든다. 포인터의 자동 분석은 극도로 어렵다. 기존 포트란 프로그램을 가지고 있고 앞으로 병렬로 처리할려고 한다면, C로 바꾸지 말라.

암시적 방법 암시적 방법은 사용자가 컴파일하기 위해 병렬화 결정의 일부(또는 모드)를 포기해야 한다. 포트란 90의 예로 고성능 포트란(High Performance FORTRAN: HPF), Bluk Bynchronous Parallel (BSP), 그리고 다른 방법의 총체적 모임들은 아직도 개발중에 있다. 암시적 방법은 사용자가 그들 풀그림의 병행성질에 대한 몇가지 정보 제공을 요구하지만, 컴파일러는 어떻게 이 병행을 병렬로 실행하는가에 대한 많은 결정을 내려야 할 것이다. 이 방법은 이동성과 효율성의 몇가지 단계를 제공하지만, 병렬 컴퓨터에 대한 병행 문제의 기술을 하는 "가장 좋은 방법"은 아직까지는 아니다.

제 5 절 베오울프 자원들

5.1 시작점

베오울프 메일링 리스트. *beowulf-request@cesdis.gsfc.nasa.gov* `<mailto:beowulf-request@cesdis.gsfc.nasa.gov>`로 메시지에 subscribe를 넣어 메일을 보내면 구독이 된다. 베오울프 홈페이지 *www.beowulf.org* `<http://www.beowulf.org>`

? Extreme Linux *www.extremelinux.org* `<http://www.extremelinux.org>`

? Extreme Linux Software from Red Hat *www.redhat.com/extreme* `<http://www.redhat.com/extreme>`

5.2 문서

베오울프 HOWTO의 최근 문서 ? The latest version of the Beowulf HOWTO *www.sci.usq.edu.au/staff/jacek/beowulf* `<http://www.sci.usq.edu.au/staff/jacek/beowulf>`

/베오울프 시스템 구성 ? Building a Beowulf System *www.cacr.caltech.edu/beowulf/tutorial/building.html* `<http://www.cacr.caltech.edu/beowulf/tutorial/building.html>`

Jacek의 베오울프 연결들 ? Jacek's Beowulf Links *www.sci.usq.edu.au/staff/jacek/beowulf* `<http://www.sci.usq.edu.au/staff/jacek/beowulf>`

베오울프 설치과 관리 HOWTO (DRAFT) ? Beowulf Installation and Administration HOWTO (DRAFT) *www.sci.usq.edu.au/staff/jacek/beowulf* `<http://www.sci.usq.edu.au/staff/jacek/beowulf>`

리눅스 병렬 처리 HOWTO ? Linux Parallel Processing HOWTO *yara.ecn.purdue.edu/˜pplinux/PPHOWTO/pphowto.html* `<http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html>`

5.3 논문

? Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Savarese, Donald Becker, and Phillip Merkey A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation. Proceedings Fifth IEEE International Symposium on High Performance Distributed Computing, 1996.

www.beowulf.org/papers/HPDC96/hpdc96.html <http://www.beowulf.org/papers/HPDC96/hpdc96.html>

? Daniel Ridge, Donald Becker, Phillip Merkey, Thomas Sterling Becker, and Phillip Merkey. Harnessing the Power of Parallelism in a Pile-of-PCs. Proceedings, IEEE Aerospace, 1997.

www.beowulf.org/papers/AA97/aa97.ps <http://www.beowulf.org/papers/AA97/aa97.ps>

? Thomas Sterling, Donald J. Becker, Daniel Savarese, Michael R. Berry, and Chance Res. Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation. Proceedings, International Parallel Processing Symposium, 1996.

www.beowulf.org/papers/IPPS96/ipps96.html <http://www.beowulf.org/papers/IPPS96/ipps96.html>

? Donald J. Becker, Thomas Sterling, Daniel Savarese, Bruce Fryxell, Kevin Olson. Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation. Proceedings,High Performance and Distributed Computing, 1995.

www.beowulf.org/papers/HPDC95/hpdc95.html <http://www.beowulf.org/papers/HPDC95/hpdc95.html>

? Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer. BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION. Proceedings, International Conference on Parallel Processing, 95.

www.beowulf.org/papers/ICPP95/icpp95.html <http://www.beowulf.org/papers/ICPP95/icpp95.html>

? 베오울프 사이트에 있는 논문들

www.beowulf.org/papers/papers.html <http://www.beowulf.org/papers/papers.html>

5.4 소프트웨어

? PVM - Parallel Virtual Machine *www.epm.ornl.gov/pvm/pvm_home.html* <http://www.epm.ornl.gov/pvm/pvm_home.html>

? LAM/MPI (Local Area Multicomputer / Message Passing Interface) *www.mpi.nd.edu/lam* <http://www.mpi.nd.edu/lam>

? BERT77 - FORTRAN conversion tool <http://www.plogic.com/bert.html> 베오울프 프로젝트 페이지에 있는 베오울프 소프트웨어 *beowulf.gsfc.nasa.gov/software/software.html* <http://beowulf.gsfc.nasa.gov/software/software.html>

Jacek의 베오울프-유틸들 ? *ftp.sci.usq.edu.au/pub/jacek/beowulf-utils* <ftp://ftp.sci.usq.edu.au/pub/jacek/beowulf-utils>

bWatch - 클러스터 감시 도구 ? bWatch - cluster monitoring tool *www.sci.usq.edu.au/staff/jacek/bWatch* <http://www.sci.usq.edu.au/staff/jacek/bWatch>

5.5 베오울프 머신들

? 아발론은 140개의 알파 프로세서, 36GB 메모리로 구성 되었고, Top 150 목록에서 144번째를 기록하며, 아마 세계에서 가장 빠른 베오울프 머신일 것이다. *swift.lanl.gov/avalon/* <http://swift.lanl.gov/avalon/>

? MPACTR은 14개의 4 CPU 페티엄 프로 200과 14GM 메모리로 이루어짐. *megalon.ca.sandia.gov/description.html* <http://megalon.ca.sandia.gov/description.html>

theHIVE은 다른 빠른 베오울프 슈퍼 컴퓨터이다. theHIVE은 64개의 노드로 128 CPU 머신과 4GB 메모리로 구성. *megalon.ca.sandia.gov/description.html* `<http://megalon.ca.sandia.gov/description.html>`

MAGI 클러스터 - 매우 흥미있는 사이트로 괜찮은 연결들이 있음. *noel.feld.cvut.cz/magi/* `<http://noel.feld.cvut.cz/magi/>`

5.6 다른 재미있는 사이트

? SMP Linux*www.linux.org.uk/SMP/title.html* `<http://www.linux.org.uk/SMP/title.html>` ? Paralogic - Buy a Beowulf*www.plogic.com* `<http://www.plogic.com>`

5.7 역사

? 전설 - 베오울프 *legends.dm.net/beowulf/index.html* `<http://legends.dm.net/beowulf/index.html>`

? 베오울프의 모험 *www.lnstar.com/literature/beowulf/beowulf.html* `<http://www.lnstar.com/literature/beowulf/beowulf.html>`

제 6 절 밑그림

6.1 sum.c

/* Jacek Radajewski *jacek@usq.edu.au* `<mailto:jacek@usq.edu.au>`*/

```
  /* 21/08/1998 */
#include <stdio.h>
  #include <math.h>
int main (void) {
  double result = 0.0;
    double number = 0.0;
    char string[80];

  while (scanf("%s", string) != EOF) {
    number = atof(string);
      result = result + number;
    }
  printf("%lf\n", result);
  return 0;
}
```

6.2 sigmasqrt.c

/* Jacek Radajewski *jacek@usq.edu.au* `<mailto:jacek@usq.edu.au>` */

```
  /* 21/08/1998 */
#include <stdio.h>
  #include <math.h>
int main (int argc, char** argv) {
  long number1, number2, counter;
    double result;
```

```
  if (argc < 3) {
      printf ("usage : %s number1 number2\n",argv[0]);
      exit(1);
    } else {
      number1 = atol (argv[1]);
      number2 = atol (argv[2]);
      result = 0.0;
    }
  for (counter = number1; counter <= number2; counter++) {
      result = result + sqrt((double)counter);
    }
  printf("%lf\n", result);
  return 0;
}
```

6.3 prun.sh

#!/bin/bash # Jacek Radajewski *jacek@usq.edu.au* `<mailto:jacek@usq.edu.au>`

```
  # 21/08/1998
export SIGMASQRT=/home/staff/jacek/beowulf/HOWTO/example1/sigmasqrt
# $OUTPUT must be a named pipe
  # mkfifo output
export OUTPUT=/home/staff/jacek/beowulf/HOWTO/example1/output
rsh scilab01 $SIGMASQRT         1  50000000 > $OUTPUT < /dev/null&
  rsh scilab02 $SIGMASQRT  50000001 100000000 > $OUTPUT < /dev/null&
  rsh scilab03 $SIGMASQRT 100000001 150000000 > $OUTPUT < /dev/null&
  rsh scilab04 $SIGMASQRT 150000001 200000000 > $OUTPUT < /dev/null&
  rsh scilab05 $SIGMASQRT 200000001 250000000 > $OUTPUT < /dev/null&
  rsh scilab06 $SIGMASQRT 250000001 300000000 > $OUTPUT < /dev/null&
  rsh scilab07 $SIGMASQRT 300000001 350000000 > $OUTPUT < /dev/null&
  rsh scilab08 $SIGMASQRT 350000001 400000000 > $OUTPUT < /dev/null&
  rsh scilab09 $SIGMASQRT 400000001 450000000 > $OUTPUT < /dev/null&
  rsh scilab10 $SIGMASQRT 450000001 500000000 > $OUTPUT < /dev/null&
  rsh scilab11 $SIGMASQRT 500000001 550000000 > $OUTPUT < /dev/null&
  rsh scilab12 $SIGMASQRT 550000001 600000000 > $OUTPUT < /dev/null&
  rsh scilab13 $SIGMASQRT 600000001 650000000 > $OUTPUT < /dev/null&
  rsh scilab14 $SIGMASQRT 650000001 700000000 > $OUTPUT < /dev/null&
  rsh scilab15 $SIGMASQRT 700000001 750000000 > $OUTPUT < /dev/null&
  rsh scilab16 $SIGMASQRT 750000001 800000000 > $OUTPUT < /dev/null&
  rsh scilab17 $SIGMASQRT 800000001 850000000 > $OUTPUT < /dev/null&
  rsh scilab18 $SIGMASQRT 850000001 900000000 > $OUTPUT < /dev/null&
  rsh scilab19 $SIGMASQRT 900000001 950000000 > $OUTPUT < /dev/null&
  rsh scilab20 $SIGMASQRT 950000001 1000000000 > $OUTPUT < /dev/null&
```