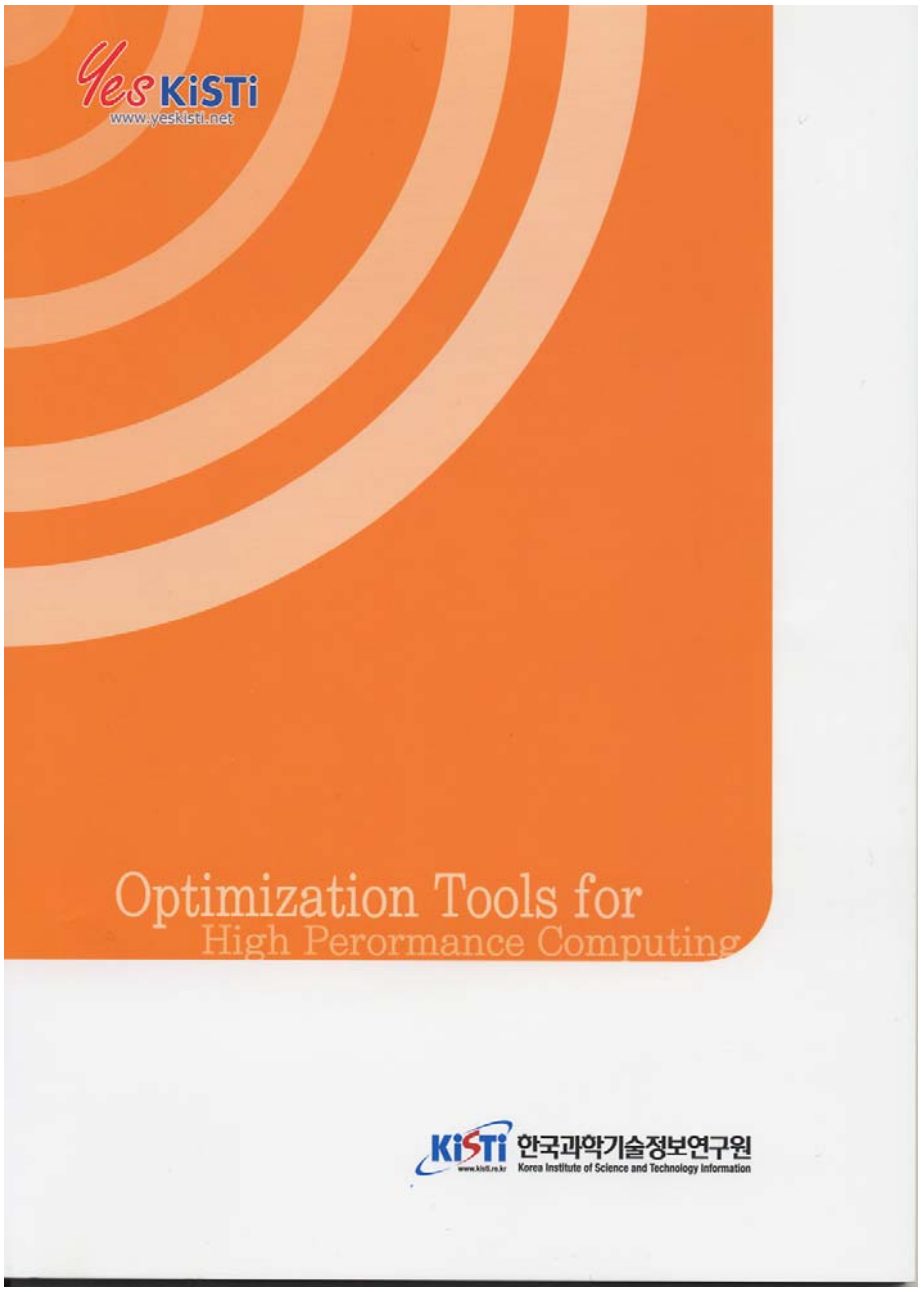
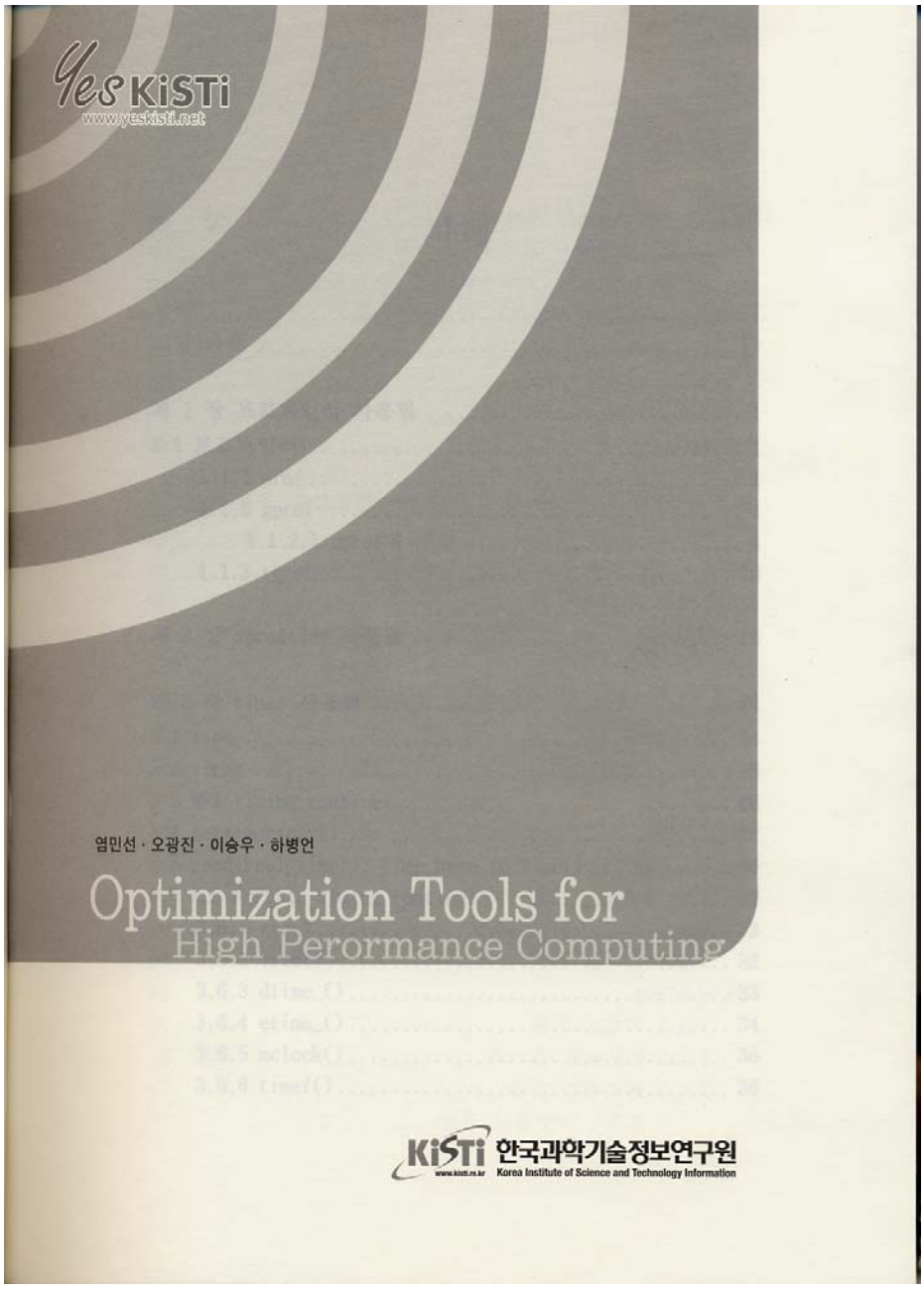

Yes KiSTi
www.yeskisti.net
Optimization Tools for
High Perormance Computing
KiSTi 한국과학기술정보연구원
Korea Institute of Science and Technology Information


Yes KiSTi
www.yeskisti.net
염민선 · 오광진 · 이승우 · 하병언
Optimization Tools for
High Perormance Computing
KiSTi 한국과학기술정보연구원
Korea Institute of Science and Technology Information

차례

그림 차례

1. 프로파일러 사용법

1.1 프로파일러

프로그램의 성능을 향상시키기 위해 많은 노력을 소비하기 전에, 프로파일러를 사용하여 성능이 얼마나 향상될 수 있는지 판별하고 최적화와 튜닝을 통해 가장 많은 성능향상을 기대할 수 있는 영역을 찾아볼 필요성이 있을 것이다. 즉 프로파일링 도구를 사용하여 프로그램의 어느 부분이 가장 자주 실행되고 어디서 대부분의 시간이 소모되는지 식별해 볼 필요성이 있다. 이번 장에서는 대표적인 프로파일링 도구인 prof, gprof, tprof의 소개와 사용법에 대해 알아 보고자 한다.

1.1.1 prof

prof 명령은 지정된 프로그램의 각 외부 루틴에 대한 CPU 사용의 프로파일을 표시하는데 자세히 설명하면 다음과 같다.

(1) 임의 루틴의 주소와 그 다음 루틴의 주소 사이에서 사용된 실행 시간의 퍼센트

(2) 함수가 호출된 횟수

(3) 단위 호출당 걸린 평균시간(ms)

prof 명령은 실행파일에 대한 monitor() 서브루틴에 의해 수집된 프로파일 데이터를 해석하여 이를 생성된 프로파일 파일(mon.out)과 관련시키며, 그 결과는 터미널로 출력되거나, 파일로 redirection할 수 있다.

prof 명령을 사용하려면 -p 옵션을 사용하여 C, FORTRAN, PASCAL, COBOL로 작성된 소스 프로그램을 컴파일한다. 이렇게 하면 monitor() 서브루틴

을 호출하는 오브젝트 파일에 프로파일링 시작함수를 삽입한다. 프로그램이 실행되면 monitor() 서브루틴이 실행 시간을 트랙하기 위한 mon.out파일을 작성한다. 또한 -p 플래그는 컴파일러가 각각의 함수에 대해 생성된 목적코드에 mcount() 서브루틴을 호출하도록 한다. 프로그램이 수행되는 동안 상위함수가 하위함수를 호출할 때 하위 함수는 상위-하위 함수의 쌍에 대해 카운터를 증가시키는 mcount() 서브루틴을 호출한다. 기본적으로 표시되는 정보는 CPU time에 대한 퍼센트에 따라 정렬되며, 이는 -t 플래그를 지정할 때와 같다.

prof 명령과 같이 사용되는 여러 가지 플래그에 대한 설명은 다음과 같다.

-a : 루틴 주소가 증가하는 방향으로 정렬

-c : 호출 회수의 감소하는 방향으로 정렬

-n : 루틴 이름에 따라 정렬

-t : 전체 계산시간에 대한 백분율이 감소하는 방향으로 정렬(기본 플래그)

-o : 루틴 이름에 따라 각 루틴의 주소를 8진법으로 표시

-x : 루틴 이름에 따라 각 루틴의 주소를 16진법으로 표시

-g : 비전역 루틴을 포함

-h : 일반적으로 표시되는 헤드를 생략

-m *MonitorData* : mon.out 파일 대신 *MonitorData* 파일로 프로파일을 할 때 사용

-z : 0번 호출되고 0초 걸린 루틴에 대해서도 프로파일하고자 할 때 사용

다음 예는 Whetstone 벤치마크 프로그램을 -p 옵션을 주어 컴파일한 후 prof 명령을 수행한 결과의 앞부분을 보여준다.

```
$ xlc -O3 -qstrict -o cwhet -p -lm cwhet.c
$ cwhet > cwhet.out
$ prof
Name                %Time     Seconds     Cumsecs  #Calls   msec/call
.main                33.6        5.92        5.92       1   5920.
.__mcount            14.9        2.62        8.54
.P3                   7.4        1.30        9.84  89900000      0.0000
.log                  7.2        1.26       11.10   9300000      0.0001
.P0                   6.8        1.19       12.29  61600000      0.0000
.exp                  6.6        1.16       13.45   9300000      0.0001
._sqrt                6.1        1.08       14.53
.cos                  6.1        1.07       15.60  19200000      0.0001
.PA                   5.3        0.93       16.53   1400000      0.0007
.atan                 3.6        0.64       17.17   6400000      0.0001
.sin                  2.5        0.44       17.61   6400000      0.0001
.__nl_langinfo_std    0.0        0.00       17.61         1      0.
.free                 0.0        0.00       17.61         2      0.
.isatty               0.0        0.00       17.61         1      0.
.__ioctl              0.0        0.00       17.61         1      0.
.ioctl                0.0        0.00       17.61         1      0.
._findbuf             0.0        0.00       17.61         1      0.
._wrtchk              0.0        0.00       17.61         1      0.
.free_y               0.0        0.00       17.61         2      0.
.exit                 0.0        0.00       17.61         1      0.
.monitor              0.0        0.00       17.61         1      0.
.moncontrol           0.0        0.00       17.61         1      0.
.printf               0.0        0.00       17.61         3      0.
```

먼저 소스 코드를 -p 옵션을 주어 컴파일을 한 후 생성된 실행파일을 실행하면 mon.out파일이 생성된다. 그런 후 prof 명령을 수행하면 실행파일과 이 mon.out 파일을 읽어 들여 프로파일을 하게 되는데 정확하게 실행하려면 $ prof cwhet -m mon.out과 같은 식으로 하면 된다.

이 예제에서 P3(), log(), P0(), exp(), cos() 루틴에 대한 호출이 여러 번 나왔다. 그래서 먼저 소스코드를 확인하여 왜 그렇게 많이 사용되었는지 확인하고, 또한 왜 이러한 루틴들의 호출을 줄일 수 있는 방안을 고려하는 것이 곧 성능향상의 길이 될 것이다.

1.1.2 gprof

gprof 명령은 C, PASCAL, FORTRAN 또는 COBOL 프로그램을 프로파일 할 때 사용된다. 이러한 gprof 명령은 프로그램이 CPU 자원을 어떻게 사용하고 있는가를 살펴볼 때 유용하게 사용되며, prof 명령보다 한 계 높은 추가적이고 가시적인 정보를 제공해 준다.

1.1.2.1 gprof의 구현

gprof 명령을 사용하려면 일단 소스 코드를 컴파일 할 때 -pg옵션을 사용하여야 한다. 이렇게 하면 컴파일러가 오브젝트 코드에 mcount() 함수의 호출을 삽입한다. mcount()함수는 상위함수가 하위함수를 호출할 때마다 카운트하고, 또한 monitor()는 각 루틴에서 걸린 시간을 측정한다.

gprof명령은 다음과 같이 2가지의 유용한 정보를 보여준다.

(1) 이 프로파일은 루틴을 CPU 시간 역순으로 그 하위 정보와 함께 보여준다. 어떤 루틴이 특정 루틴을 가장 많이 호출했고 특정 루틴에 의해 어떤 하위 루틴이 가장 자주 호출되었는지를 알 수 있다.

(2) prof 명령이 보여주는 정보와 유사하게 CPU 사용의 플랫 프로파일을 보여주며 이는 또한 루틴의 사용 및 호출회수에 따른 정보를 보여 준다.

gprof 명령과 같이 유용하게 사용되는 플래그에 대한 설명은 다음과 같다.

-b : 프로파일의 각 열에 있는 설명을 출력하지 않음

-e *Name* : *Name* 루틴과 모든 그 하위 루틴에 대한 프로파일을 출력하지 않음

-f *Name* : *Name* 루틴과 모든 그 하위 루틴에 대한 프로파일을 출력

소스코드를 -pg 옵션을 주어 컴파일한 후 실행을 하면 gmon.out 파일이

생성이 되는데, 이 파일에는 다음과 같은 정보가 binary형태로 저장되어 있다.

- 실행파일 및 공유 라이브러리 오브젝트 이름
- 각 프로그램 세그먼트에 할당된 가상 메모리 주소
- 각 상위-하위 루틴에 대한 mcount() 데이터
- 각 프로그램 세그먼트에 대해 누적된 시간(ms)

gprof 명령이 실행되면 실행파일과 gmon.out 파일을 함께 읽어 들여 call-graph 프로파일과 플랫 프로파일을 생성한다. 일반적으로 gprof 명령의 출력 내용이 아주 길기 때문에 임의의 파일로 redirection하여 저장한 후 열어보는 것이 좋다.

다음 예는 Whetstone 벤치마크 프로그램을 gprof 명령을 통해 프로파일하는 과정을 보여 준다.

```
$ xlc -O3 -qstrict -o cwhet -pg -lm cwhet.c
$ cwhet > cwhet.out
$ gprof cwhet > cwhet.gprof
```

앞서 설명한 것처럼 소스코드를 -pg 옵션을 주어 컴파일한 후 실행하면 gmon.out 파일이 생성됨을 확인 할 수 있다. 그런 후 위 예의 마지막 부분과 같이 실행하면 cwhet.gprof파일에 gprof명령의 결과들이 redirection되어 저장된다. cwhet.gprof 파일의 내용은 call-graph 프로파일, 플랫 프로파일 그리고 함수 이름 인덱스로 나눌 수 있는데 각각에 대해서 자세히 살펴보면 다음과 같다.

Call-Graph 프로파일

call-graph 프로파일은 cwhet.gprof 파일의 처음 부분이며 다음과 같은 형식이다.

```
The sum of self and descendents is the major sort
         for this listing.

         function entries:

index    the index of the function in the call graph
         listing, as an aid to locating it (see below).

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

처음 부분은 각 열에 대한 설명을 나타내는 것으로 이는 앞에서 설명한 바와 같이 gprof 명령을 수행할 때 -b 플래그를 주어 실행하면 표시되지 않는다.

```
ngranularity: Each sample hit covers 4 bytes. Time: 21.09 seconds

                                called/total       parents
index  %time    self descendents  called+self    name           index
                                called/total       children

                6.30        8.12       1/1           .__start [2]
[1]     68.4    6.30        8.12       1         .main [1]
                2.30        0.00 89900000/89900000     .P3 [4]
                1.33        0.00 9300000/9300000     .exp [5]
                1.03        0.00 9300000/9300000     .log [7]
                0.93        0.00 19200000/19200000     .cos [8]
                0.91        0.00 1400000/1400000     .PA [9]
                0.74        0.00 6400000/6400000     .atan [10]
                0.54        0.00 61600000/61600000     .P0 [11]
                0.34        0.00 6400000/6400000     .sin [12]
                0.00        0.00       3/3           .printf [22]
                0.00        0.00       2/2           .time [27]

-----------------------------------------------

6.6s                                                 <spontaneous>
[2]     68.4    0.00       14.42                 .__start [2]
                6.30        8.12       1/1           .main [1]
                0.00        0.00       1/1           .exit [33]
```

```
-----------------------------------------------

6.6s                                              <spontaneous>
[3]     23.7    4.99        0.00                  .__mcount [3]

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

다음에는 각 함수에 대한 프로파일 정보가 보여지는데 이를 읽는 방법은 다음과 같다. 먼저 첫 번째 인덱스가 [1]이라고 표시 되어 있는데 이렇게 표시되어 있는 부분이 현재 함수를 나타낸다. 즉 .main함수가 현재 함수이고 이는 .__start[2]라는 상위함수가 호출을 하여 시작되었으며, 그 밑에 있는 .P3[4], .exp[5], .log[7] 등의 하위함수들을 호출하고 있다. 하위함수들은 위와 같이 현재함수의 아래에 위치해 있다. 현재함수에 대한 하위함수의 self열과 descendents 열에 있는 시간들이 모두 더해져서 현재함수의 descendents에 표시된다. 즉 위의 예에서는 하위함수인 .P3[4] ~ .time[27]함수들의 self 열과 descendents 열에 있는 시간들을 모두 더하면 .main[1]의 descendents에 표시되어 있는 8.12초가 된다.

그 아래에는 첫 번째 인덱스가 [2]라고 표시되어 있으며 이는 곧 .__start[2] 함수가 현재함수이면서 .main[1], .exit[33]이라는 하위함수를 호출하고 있다.

이와 같이 gprof 명령에 의해 수행한 프로그램의 함수들에 대한 상위-하위관계 즉 트리 구조를 일반 텍스트로 알 수 있으며 이러한 구조 내에서 각 함수들의 수행시간을 알 수 있다.

플랫 프로파일

플랫 프로파일은 prof 명령에 의한 결과와 상당히 유사하며, prof 명령에 의해 보여지는 정보에 몇가지가 추가되어 있다.

```
flat profile:

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ngranularity: Each sample hit covers 4 bytes. Time: 21.09 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 29.9      6.30     6.30        1  6300.00 14420.00  .main [1]
 23.7     11.29     4.99                             .__mcount [3]
 10.9     13.59     2.30 89900000     0.00     0.00  .P3 [4]
  6.3     14.92     1.33  9300000     0.00     0.00  .exp [5]
  5.2     16.02     1.10                             ._sqrt [6]
  4.9     17.05     1.03  9300000     0.00     0.00  .log [7]
  4.4     17.98     0.93 19200000     0.00     0.00  .cos [8]
  4.3     18.89     0.91  1400000     0.00     0.00  .PA [9]
  3.5     19.63     0.74  6400000     0.00     0.00  .atan [10]
  2.6     20.17     0.54 61600000     0.00     0.00  .P0 [11]
  1.6     20.51     0.34  6400000     0.00     0.00  .sin [12]
  1.2     20.77     0.26                             .__stack_pointer [13]
  0.9     20.96     0.19                             .qincrement1 [14]
  0.6     21.09     0.13                             .qincrement [15]
  0.0     21.09     0.00       11     0.00     0.00  .fwrite [16]
  0.0     21.09     0.00       11     0.00     0.00  .fwrite_unlocked [17]
  0.0     21.09     0.00       11     0.00     0.00  .memchr [18]
  0.0     21.09     0.00        3     0.00     0.00  ._doprnt [19]
  0.0     21.09     0.00        3     0.00     0.00  ._xflsbuf [20]
  0.0     21.09     0.00        3     0.00     0.00  ._xwrite [21]

  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

이는 Call-Graph 프로파일 다음 부분이며, 마찬가지로 각 열에 대한 설명이 필요 없을 경우에는 gprof 명령을 수행할 때 -b 플래그를 주어 실행하면 된다. 여기서 보여지는 프로파일 결과 중에서 self seconds 열은 하위함수에서 걸린 시간은 제외한 단지 그 함수에서만 사용된 CPU 시간(초)을 의미하며, calls 열은 각 함수가 상위함수에 의해 호출된 회수를 의미한다.

일반적으로 목록의 위쪽에 있는 함수들이 최적화를 할 때 먼저 고려해야 할 대상이지만, 경우에 따라서는 이러한 함수들이 얼마나 많이 호출이 되었는지를 파악해야 한다. 즉 자주 호출되는 함수를 조금 향상시키는 것이 최적화에 있어서 아주 많을 영향을 끼칠 수 있다.

함수 이름 인덱스

gprof 명령의 마지막 부분에는 그 위에서 나온 함수들의 이름과 인덱스를 정리해 두었다. 다음과 같은 형식으로 정리되어 있으며, 함수 이름 순으로 정렬되어 있다.

```
Index by function name

 [11] .P0                  [8] .cos                 [38] .monitor
  [4] .P3                 [33] .exit                [39] .myfcvt
  [9] .PA                  [5] .exp                 [40] .nl_langinfo
 [29] .__ioctl            [25] .free                [41] .pre_ioctl
  [3] .__mcount           [26] .free_y              [22] .printf
 [30] .__nl_langinfo_std  [16] .fwrite              [15] .qincrement
 [13] .__stack_pointer    [17] .fwrite_unlocked     [14] .qincrement1
 [19] ._doprnt            [34] .ioctl               [12] .sin
 [31] ._findbuf           [35] .isatty              [23] .splay
  [6] ._sqrt               [7] .log                 [27] .time
 [32] ._wrtchk            [1] .main                 [28] .time_base_to_time
 [20] ._xflsbuf           [18] .memchr              [24] .write
 [21] ._xwrite            [36] .mf2x1
 [10] .atan               [37] .moncontrol
```

1.1.3 tprof

tprof 명령은 프로그램과 시스템 전체에 대한 CPU 사용량에 대한 정보를 보여준다. 이 명령은 프로그램이 어느 부분에서 CPU 사용이 가장 많은지를 알고자 하는 JAVA, C, C++, FORTRAN 프로그래머들에게 매우 유용한 도구이다. 또한 이 명령은 유휴 CPU 시간에 대한 비율도 보여 주며, 이러한 정보들은 전체적인 CPU 이용률을 파악하는데 효율적으로 이용된다. 프로파일하고자 하는 프로그램을 지정하여 그 프로그램을 실행시키면 tprof 명령이 프로파일 결과를 포함하는 파일들을 생성시킨다. CPU 시간을 소스 코드의 행과 연관시키는 것을 마이크로프로파일링이라고 하는데 tprof 명령을 이용하여 마이크로프로파일링을 수행하기 위해서는 소스 코드를 -g 옵션을 주어 컴파일하여야 하며, 또한 tprof 명령을 수행하는 디렉토리에 소스 코드가 있어야 한다.

다음 C 프로그램을 tprof를 이용하여 마이크로프로파일링하는 과정을 보도록 하겠다.

```
$ cat version1.c
#include <stdlib.h>
#define Asize 4096
#define RowDim InnerIndex
#define ColDim OuterIndex
main()
{
        int Increment;
        int OuterIndex;
        int InnerIndex;
        int big [Asize][Asize];
        /* Initialize every byte of the array to 0x01 */
        for (OuterIndex=0; OuterIndex<Asize; OuterIndex++)
        {
                for (InnerIndex=0; InnerIndex<Asize; InnerIndex++)
                        big [RowDim][ColDim] = 0x01010101;
        }
        Increment = rand();
        /* Increment every element in the array */
        for (OuterIndex=0; OuterIndex<Asize; OuterIndex++)
        {
                for (InnerIndex=0; InnerIndex<Asize; InnerIndex++)
                {
                        big [RowDim][ColDim] += Increment;
                        if (big [RowDim][ColDim] <0)
                        printf("negative number. %d₩n", big[RowDim][ColDim]);
                }
        }
        printf("version 1 check num: %d₩n", big[rand()%Asize][rand()%Asize]);
        return(0);
}
```

위의 version1.c 코드를 -g 옵션을 주어 컴파일한 후 다음과 같이 tprof 명령을 실행하면 프로파일 정보가 포함되어 있는 여러 파일들이 생성된다. 생성된 파일들은 모드 __(두개의 밑줄)로 시작하는 파일들이며 기존에 있는 파일들과 쉽게 구별이 가능하다.

```
$ ls
```

```
version1.c
$ xlc -O2 -g -o version1 version1.c
$ ls
version1    version1.c
$ tprof -mp version1 -x version1
Starting Trace now
Starting  version1
Wed May  7 15:14:42 2003
System: AIX ***** Node: 5 Machine: ***********
version 1 check num: 16859847
Trace is done now
 * Samples from __trc_rpt2
 * Reached second section of __trc_rpt2
$ ls
__h.version1.c       __tmp.j              __tmp.s              __version1.all
__ldmap              __tmp.k              __tmp.u              version1
__t.main_version1.c  __tmp.o              __trc_rpt2           version1.c
```

생성된 여러 파일들 중에서 __version1.all, __t.main_version1.c, __h.version1.c 파일이 중요한 프로파일 정보들을 담고 있는데 먼저 __version1.all 파일의 내용을 살펴 보면 다음과 같다.

```
$ cat __version1.all

         Process           PID         TID    Total       Kernel       User
Shared   Other
        =======      ===      ===  ===== ======   ==== ======  =====
         version1       3350614    2580671   831       15        816       0
0
        =======      ===      ===  ===== ======   ==== ======  =====
          Total                              831       15        816       0
0

           Process     FREQ   Total    Kernel      User  Shared     Other
          =======    ===  ===== ======   ==== ======  =====
           version1       1      831     15         816     0         0
          =======    ===  ===== ======   ==== ======  =====
            Total          1     831      15        816     0         0

 Total System Ticks: 3351 (used to calculate function level CPU)
```

```
  Total Ticks For version1 (USER) = 816

 Subroutine                          Ticks     %      Source          Address
Bytes
 =============              ===== ==== ========   ========== ======
 .main                                816     24.4    version1.c          350
170
```

위에서 첫 부분은 프로세스의 이름, PID 등과 함께 소요된 tick의 수를 보여 준다. 여기서 tick은 초당 100번의 비율로 일어나며 하나의 tick은 1/100초 즉 10 msec를 의미한다. version1 프로세서는 커널 영역에서 15번, 사용자 영역에서 816번의 tick이 일어나서 총 831번의 tick이 일어나며 이는 곧 커널 영역에서 0.15초, 사용자 영역에서 8.16초의 시간을 의미한다. 두 번째 부분은 각 프로그램을 수행한 다른 여러 프로세스의 수(FREQ)를 나타내며, 세 번째 부분은 실행파일의 각 함수들에 의해 소요된 tick의 수와 각 함수에 대한 CPU tick의 퍼센트를 나타낸다.

다음으로 __t.main_version1.c 파일에 대해서 살펴보면 이는 main에 대한 프로파일된 소스를 포함하고 있다.

```
$ cat __t.main_version1.c

 Ticks Profile for main in ./version1.c

   Line   Ticks   Source

    13      -          for (OuterIndex=0; OuterIndex<Asize; OuterIndex++)
    14      -          {
    15       -                         for (InnerIndex=0; InnerIndex<Asize;
InnerIndex++)
    16    298                          big [RowDim][ColDim] = 0x01010101;
    17      -          }
    18      -          Increment = rand();
    19      -          /* Increment every element in the array */
    20      -          for (OuterIndex=0; OuterIndex<Asize; OuterIndex++)
    21      -          {
    22       -                         for (InnerIndex=0; InnerIndex<Asize;
InnerIndex++)
    23      -                 {
    24      1                          big [RowDim][ColDim] += Increment;
```

```
    25    517                          if (big [RowDim][ColDim] <0)
    26      -                               printf("negative number. %d₩n",
big[RowDim][ColDim]);
    27      -                   }
    28      -           }
    29          -                   printf("version 1 check num: %d₩n",
big[rand()%Asize][rand()%Asize]);
    30      -           return(0);
    31      -     }

 816 Total Ticks for main in ./version1.c
```

위와 같이 main 코드에서 코드의 각 줄에서 tick이 얼마나 소요되었는지를 알 수 있으며 보이는 것처럼 16번째 줄에서 298번, 24번째 줄에서 1번 25번째 줄에서 517번의 tick이 일어 났으며, 이는 곳 16번째 줄에서 2.98초, 24번째 줄에서 0.01초 25번째 줄에서 5.17초의 시간이 걸렸다는 것을 알 수 있다.

다음은 __h.version1.c 파일로 이는 소스 코드에서 가장 자주 사용되는 행의 번호를 나타낸다.

```
$ cat __h.version1.c

 Hot Line Profile for ./version1.c

   Line   Ticks

    25    517
    16    298
    24      1
```

즉 __t.main_version1.c 파일이 포함하는 내용을 요약하여 가장 많이 사용된 tick의 수에 따라 코드에서 줄의 번호를 나열하였다.

이와 같이 tprof 명령의 최대 장점이라면 이 명령을 이용한 마이크로프로파일링을 통하여 소스 코드 레벨에서의 프로파일 정보를 알 수 있다는 것이다. 그래서 소스 코드내에서의 핫스팟을 파악하여 그러한 부분을 집중

적으로 최적화함으로써 성능 향상을 기대할 수 있다.

2. xprofiler 사용법

프로그램을 컴파일할 때 -pg옵션을 주고 컴파일하여 실행을 하면 xprofiler를 이용하여 프로그램의 어떤 서브루틴에서 시간이 얼마나 걸렸으며, 각 서브루틴들에서 걸린시간이 전체 계산시간에서 어떻게 분포가 되는지 등에 대한 여러가지 정보들을 그래픽 환경에서 아주 편하고 쉽게 확인해 볼수 있다.

실행파일이 md_systolic.x이고 실행시킨 후 생성된 파일이 gmon.out.0이라고 할 때 다음과 같이 실행하면 된다.(여기서 gmon.out파일은 Serial 프로그램에서는 gmon.out파일 하나만 생성이 되고 parallel 프로그램에서는 각 pid에 해당하는 번호가 gmon.out.뒤에 붙는데 이때에는 각 pid에 해당것 전체를 프로파일링해 볼 수 있다.)

```
$ xprofiler md_systolic.x gmon.out.0
```

그러면 다음과 같은 화면을 볼 수 있다.

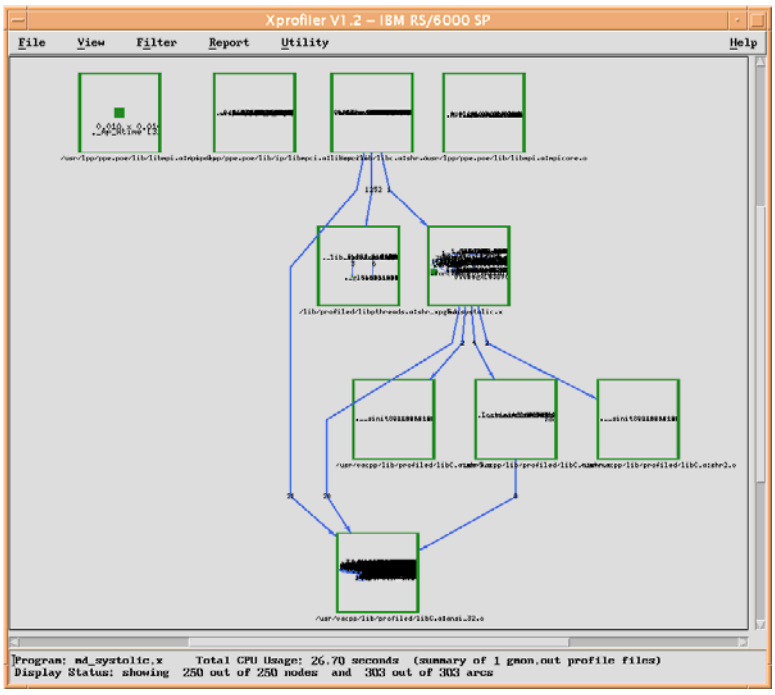


그림 2.1 xprofiler 실행화면

여기서 library에 해당되는 부분을 숨길려면 메뉴에서 Filter -> Hide All Library Calls를 클릭하면 된다.

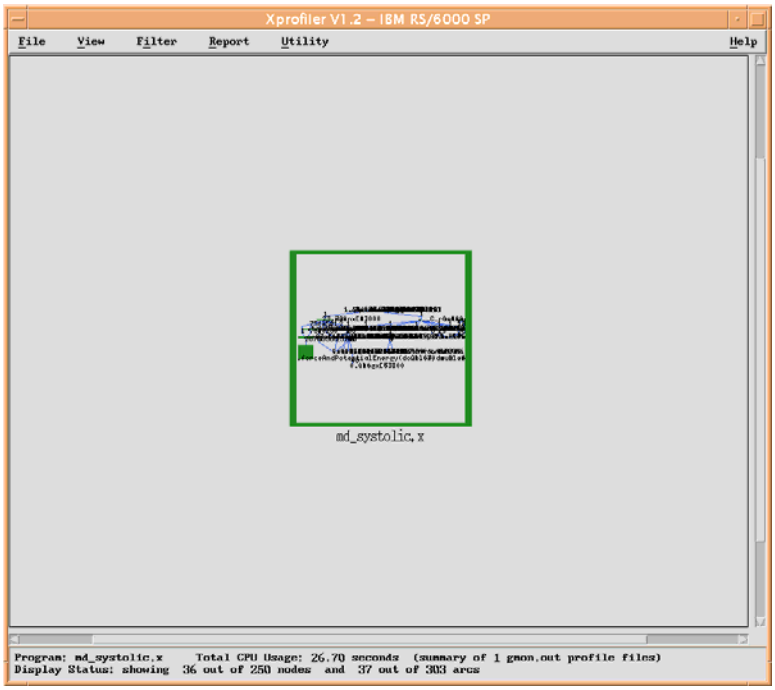
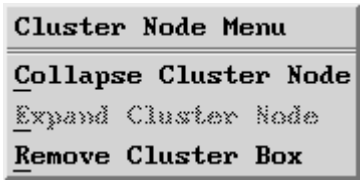


그림 2.2 xprofiler 세부 실행화면

위의 그림에서 녹색의 사각 박스의 회색부분에 오른쪽 마우스를 클릭하면 다음과 같은 메뉴가 나타난다.



여기서 Remove Cluster Box를 클릭하면 다음과 같이 md_systolic.x에 해당되는 부분이 확대 된다.

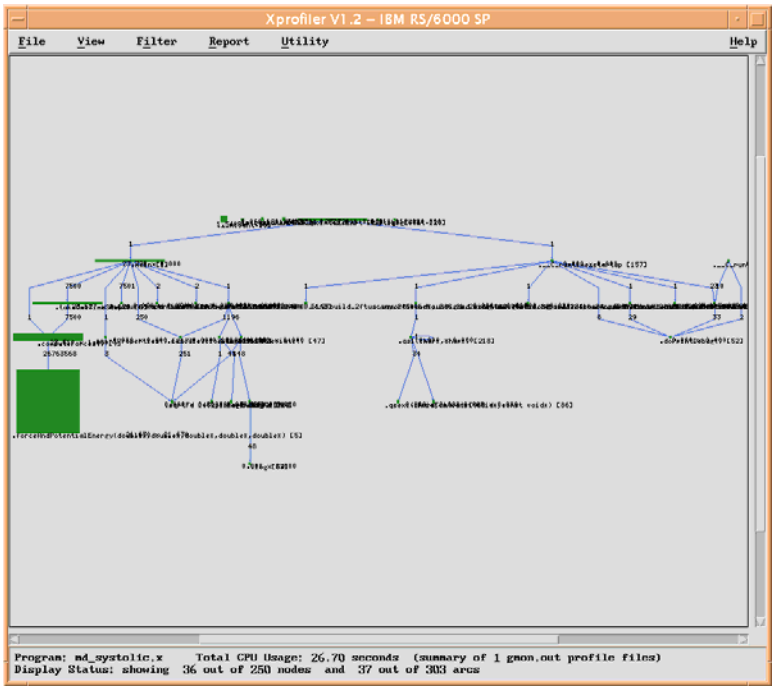
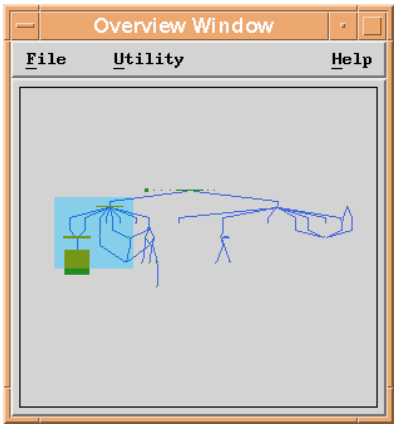


그림 2.3 xprofiler 세부 실행화면2

이제 찾고자하는 부분을 좀더 확대해서 볼려면 메뉴에서 View -> Overview를 클릭하여 다음과 같이 Overview Window를 띄운 다음 파란색의 사각형을 작게 축소하면 그 부분에 해당되는 부분을 전체 화면에서 자세히 확인해 볼 수 있다.


Overview Window
File
Utility
Help

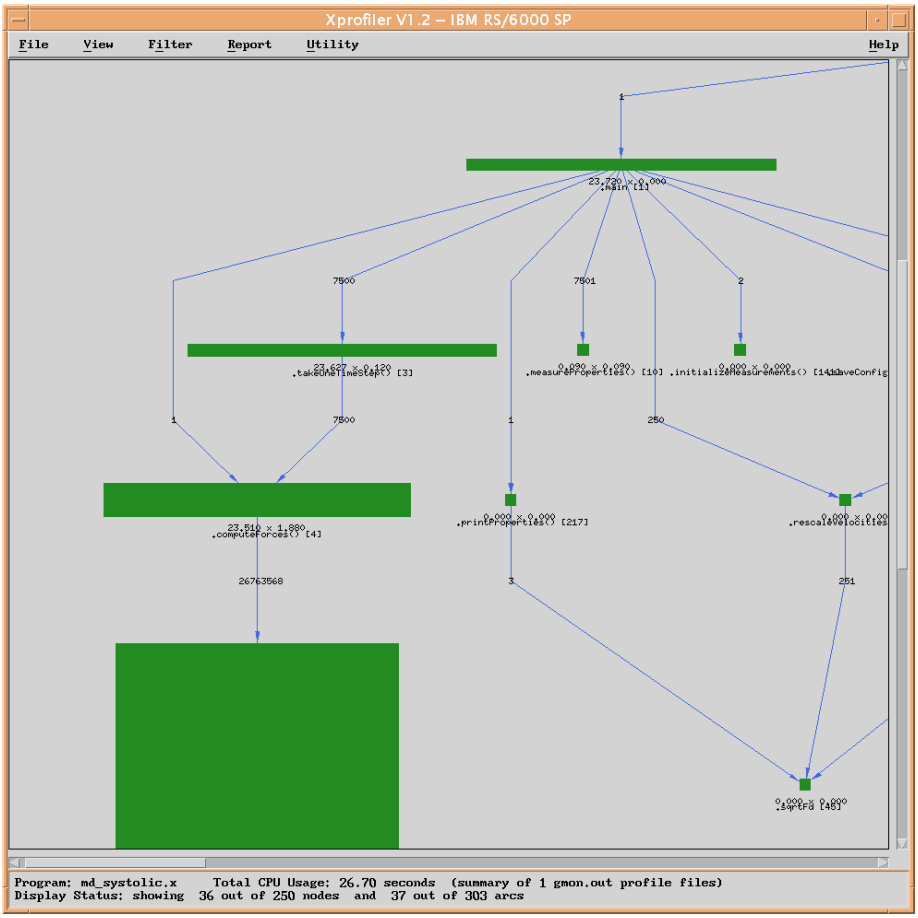


그림 2.4 xprofiler 세부 실행화면3

위의 그림에서 크고 작은 크기의 녹색 사각형이 나타나는데 이 녹색 사각형 바로 밑에 A x B 형태로 계산 시간이 적혀 있고, 다시 그 밑에 서브루틴의 이름이 적혀 있다. A x B 형태로 적힌 계산 시간에서 B 해당하는 시간은 순수하게 그 서브루틴에서 걸린 시간을 나타내고 A에 해당하는 시간은 하위서브루틴에서 걸린 시간과 B에 해당하는 시간을 합친 그 루틴의 전체 계산시간을 나타낸다. 또한 녹색 사각형에서 가로의 길이는 위의 A에 해당하는 시간을 나타내고 세로의 길이는 위의 B에 해당하는 시간을 나타낸다.

Serial 코드를 병렬화할 때 위와 같이 프로파일링을 한 후 녹색으로 표시된 사각형의 면적이 넓은 곳을 찾아서 효율적으로 병렬화를 할 수 있다.

또한 메뉴에서 Report를 클릭하여 여러가지 다양한 정보를 확인해 볼수 있으며 특히 Report -> Flat profile를 이용하여 위의 그림에 해당하는 내용을 표로 확인해 볼 수 있다.

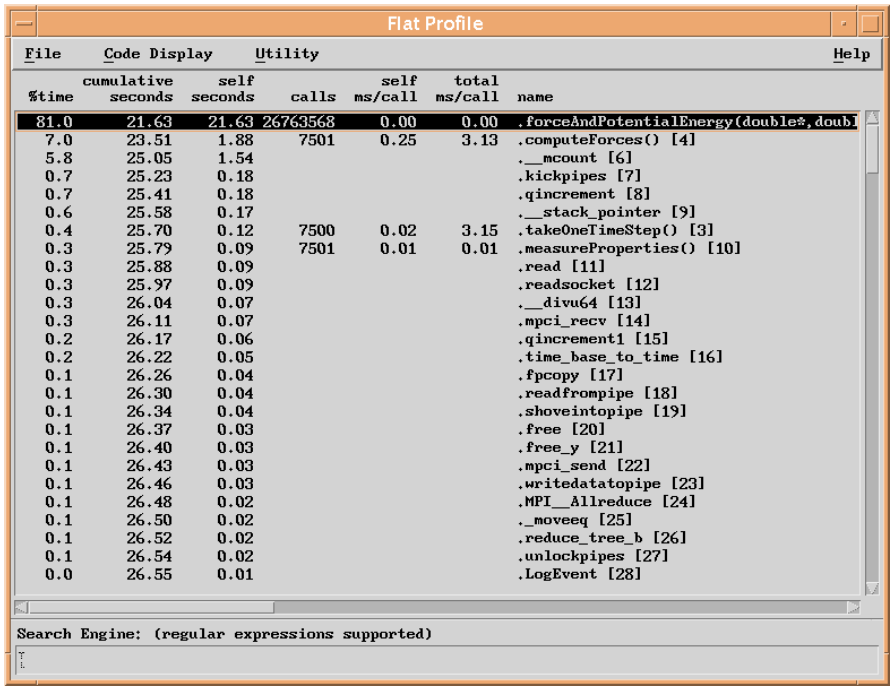


그림 2.5 Flat Profile

위의 메뉴에서 Code Display -> Show Source Code 클릭하여 다음과 같이 위에서 클릭한 서브루틴의 소스코드를 확인해 볼 수 있다.

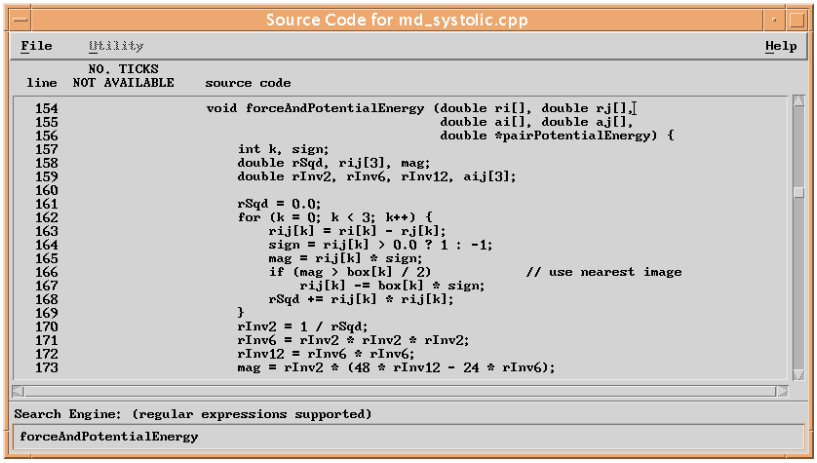


그림 2.6 Source Code

3. timer 사용법

3.1 time

time command는 유닉스 shell command로 프로그램의 총 실행시간을 반환한다. 출력 형식은 korn shell과 c shell에서 조금씩 다르며 기본적으로 real time, user time, system time의 세가지 정보를 출력한다.

① real time : 프로그램이 시작되어 마칠 때까지 걸리는 실제 총 실행 시간(wall-clock time)
② user time : 프로그램 실행에 사용된 총 CPU 시간
③ system time : 프로그램 실행에서 운영체제 호출에 사용된 총 CPU 시간

Korn shell에서 time command의 실행 결과

```
$ time mpitest -procs 4
real    0m2.87s
user    0m1.29s
sys     0m1.57s
```

C shell에서 time command의 실행 결과

```
$ time mpitest -procs 4
1.150u 0.020s 0:01.76 66.4% 15+3981k 24+10io 0pf+0w
  ①    ②      ③    ④      ⑤       ⑥    ⑦  ⑧
```

C shell에서의 실행결과는 각각 다음의 의미를 가진다.

① user CPU time : 1.15초
② system CPU time : 0.02초
③ real time(wall-clock time) : 0분 1.76초
④ real time에서 CPU time(user+system)이 차지하는 정도 : 66.4%

⑤ 메모리 사용량 : 공유(15Kbytes) + 공유되지 않음(3981Kbytes)
⑥ 입력(24블록) + 출력(10블록)
⑦ 페이지 폴트 없음
⑧ 스왑 없음

3.2 timex

timex command는 IBM Unix인 AIX에서 사용 가능한 명령어로 별도의 옵션 없이 사용하면 어떤 shell 환경에서든지 time 명령어를 korn shell에서 사용한 것과 똑 같은 형식의 결과를 출력한다. timex 명령어는 여러 가지 옵션과 같이 사용하여 프로세스 통계 등과 같이 다양한 결과를 출력할 수 있다.
-o : 읽혀지거나 기록된 전체 블록 수와 실행 프로그램과 그 하위 프로세스가 전송한 전체 문자 수를 출력한다.
-p : 실행 프로그램과 그 하위 프로세스에 대한 프로세스 사용 통계 레코드를 출력한다.
-s : 프로그램 실행 중 전체 시스템 활동(sar 명령에 나열된 모든 데이타 항목)을 출력한다.

timex command의 실행결과(-p 옵션)

```
$ timex -p naive
real 64.66
user 64.63
sys  0.02

START AFT: Wed Jan 29 16:49:49 KORST 2003
END BEFOR: Wed Jan 29 16:50:54 KORST 2003
COMMAND                    START  END     REAL    CPU     CHARS    BLOCKS
NAME      USER  TTYNAME  TIME   TIME    (SECS)  (SECS)  TRNSFD    READ
naive      my_id   pts/52  16:49:49 16:50:53  64.64    64.64   4000256     0
```

3.3 MPI timing routines

MPI는 고유의 timing 루틴을 가지고 있다. MPI 프로그램 내에서 특정 영역의 실행시간을 측정하기 위해 사용되는 MPI_Wtime()과 MPI_Wtick()을 소개한다.

① MPI_Wtime() : 루틴을 호출하는 현재 time의 값을 초 단위의 double precision 실수로 출력한다. 출력 값은 과거 어느 시점으로부터의 경과 시간을 나타내며 두 지점에서 출력된 time 값의 차이가 호출된 두 지점사이의 실제 실행시간을 나타내게 된다.

② MPI_Wtick() : MPI_Wtime()의 호출에 의해 출력되는 time의 정밀도를 초단위로 출력한다. 출력되는 값은 연속되는 시간 간격을 초단위로 나타낸 것이다.

MPI timing 루틴의 사용에 사용된 예제(Fortran)와 실행결과

```
program wtime_test
include 'mpif.h'
integer n,m,ierr
double precision start,end,resolution
call MPI_INIT(ierr)
start = MPI_WTIME()
do m=1,2000000
   n = n + m
end do
end = MPI_WTIME()
resolution = MPI_WTICK()
print *,'Wallclock times(secs): start= ',start,'end=',end
print *,'elapsed=',end-start,'resolution=',resolution
call MPI_FINALIZE (ierr)
end
```

```
$ wtime_test
 Wallclock times(secs): start=  1043828035.01158202 end= 1043828035.04210198
 elapsed= 0.3051996231079101 56E-01 resolution= 0.16199999999999995E-06
```

3.4 gettimeofday()

gettimeofday() 루틴은 대부분의 Unix 시스템 표준 C 라이브러리에 포함되어 있으며, 1970년 1월 1일 0시부터의 시간을 초와 백만분의 일초 단위로 출력한다. C 프로그램의 어디에서든 호출하여 코드 부분의 시작시간과 종료시간을 알 수 있으며 종료시간과 시작시간의 차이를 구해 코드 부분의 실행시간을 알아볼 수 있다.
측정 시간의 정밀도는 시스템마다 다르지만 IBM 시스템에서는 백만분의 일초 단위의 정밀도를 가진다.

gettimeofday() 루틴의 사용 예제(C)와 실행결과

```
/* gettime.c */
#include <stdio.h>
#include <sys/time.h>
#define ARRAY_SIZE        1000

float a[ARRAY_SIZE][ARRAY_SIZE];
float b[ARRAY_SIZE][ARRAY_SIZE];
float c[ARRAY_SIZE][ARRAY_SIZE];
struct timeval start_time, end_time;

main()
{
   int i, j;
   int total_usecs;

   /* First, call gettimeofday() to get start time */
   gettimeofday(&start_time, (struct timeval*)0);

    for(i=0;i<1000;i++)
       for(j=0;j<1000;j++)
          c[i][j] = c[i][j] + a[i][j] * b[i][j];

   /* Now call gettimeofday() to get end time */
   gettimeofday(&end_time, (struct timeval*)0);  /* after time */

   /* Print the execution time */
   total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
                 (end_time.tv_usec-start_time.tv_usec);

   printf("Total time was %d uSec.₩n", total_usecs);
}
```

```
$ gettime
 Total time was 48616 uSec.
```

위의 예제에서 결과값은 예제내의 코드 부분의 실행시간이 48616(μs)임을 보여준다. Fortran에서 C 라이브러리에 포함된 gettimeofday() 루틴을 사용하기 위해서는 언어간 호출(inter-language call)을 사용해야 하는

데 간단한 사용법은 다음과 같다. Fortran에서 C 함수를 호출하기 위해 우선 아래와 같은 wrapper 프로그램을 작성하고 cc -c wrap_gettime.c 와 같이 컴파일하여 목적코드 wrap_gettime.o를 생성한다.

Fortran에서 gettimeofday()를 호출하기위한 wrapper 프로그램

```
/* wrap_gettime */
#include <stdio.h>
    #include <sys/time.h>
    void timing_fgettod (int times[2])
    {
      struct timeval tv;
      gettimeofday(&tv, (struct timeval*)0);
      times[0] = tv.tv_sec;
      times[1] = tv.tv_usec;
    }
```

Fortran 프로그램에서 external 함수 wrap_gettime를 호출하며, xlf -o gettime gettime.f wrap_gettime.o 와 같이 컴파일하여 사용한다.

gettimeofday() 루틴의 사용 예제(Fortran)와 실행결과

```
      program gettime
      external wrap_gettime
      integer ARRAY_SIZE, SEED
      parameter(ARRAY_SIZE=1000)

      real*8 a(ARRAY_SIZE,ARRAY_SIZE)
      real*8 b(ARRAY_SIZE,ARRAY_SIZE)
      real*8 c(ARRAY_SIZE,ARRAY_SIZE)
      integer times1(2), times2(2)
      integer i, j, total_usecs

C     First, call wrap_gettime to get the start time
      call wrap_gettime (%REF(times1))

      do i=1,1000
        do j=1,1000
```

```
          c(i,j) = c(i,j) + a(i,j) * b(i,j)
        end do
      end do

C     Now call wrap_gettime to get the end time
      call wrap_gettime (%REF(times2))

C     Print the execution time
      total_usecs = ((times2(1) - times1(1))*1000000) + times2(2) -
     &              times1(2)
      print *, 'Total time was ',total_usecs, 'uSec'
      end
```

```
$ gettime
 Total time was 323142 uSec.
```

3.5 read_real_time(), time_base_to_time()

read_real_time()과 time_base_to_time()은 모두 IBM AIX 시스템의 표준 C 라이브러리에 포함된 timer 루틴들이다. 두 루틴은 보다 고 정밀도의 경과 시간 측정을 위해 사용되며, 10억분의 일초 단위의 정밀도를 가지는 경과 시간을 출력한다.

read_real_time()은 프로세서 리얼 타임 클락 또는 타임 레지스터를 읽어 들이고, time_base_to_time()은 read_real_time()이 읽어 들인 값을 초와 10억분의 일초 단위의 리얼타임으로 변환시키는 역할을 한다.

read_real_time()과 time_base_to_time() 루틴의 사용 예제(C)와 실행결과

```
/* nanotime.c */
#include <stdio.h>
#include <sys/time.h>
```

```
#define ARRAY_SIZE 10000

float a[ARRAY_SIZE][ARRAY_SIZE];
float b[ARRAY_SIZE][ARRAY_SIZE];
float c[ARRAY_SIZE][ARRAY_SIZE];

main(void)
{
   timebasestruct_t start, finish;
   int secs, n_secs;
   int i, j;

   /* get the time before the operation begins */
   read_real_time(&start, TIMEBASE_SZ);
   for(i=0;i<10000;i++)
       for(j=0;j<10000;j++)
          c[i][j] = c[i][j] + a[i][j] * b[i][j];

   /* get the time after the operation is complete */
   read_real_time(&finish, TIMEBASE_SZ);

   /* Call the conversion routines */
    time_base_to_time(&start, TIMEBASE_SZ);
    time_base_to_time(&finish, TIMEBASE_SZ);

   /* subtract the starting time from the ending time */
   secs = finish.tb_high - start.tb_high;
   n_secs = finish.tb_low - start.tb_low;

   (void) printf("Sample time was %d seconds %d nanoseconds₩n",
                secs, n_secs);

   exit(0);
}
```

```
$ nanotime
 Sample time was 4 seconds 372269354 nanoseconds
```

3.6 XLF Fortran timing routines

rtc(), irtc(), dtime_(), etime_(), mclock(), timef() 등은 IBM의 XLF Fortran 라이브러리에 포함된 timer루틴들이다.

3.6.1 rtc()

함수 rtc()는 사용이 간편하고 백만분이 일초 단위의 정밀도를 가지는 Real(8) 값을 출력한다.

rtc()함수의 사용 예제(Fortran)와 실행결과

```
   PROGRAM RTC_TIME
   INTEGER SIZE
   PARAMETER(SIZE=5000)
   REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
   REAL(8) A, B, rtc
   A = rtc()
   DO i = 1, SIZE
      DO j = 1, SIZE
         N(i,j) = N(i,j) + M(i,j)*L(i,j)
      ENDDO
   END DO
   B = rtc()
   PRINT *, 'Seconds elapsed: ',B - A
   END
```

```
$ rtc_time
 Seconds elapsed : 20.0878545045852661
```

3.6.2 irtc()

irtc() 함수는 rtc()와 똑 같은 방법으로 사용하지만 10억분의 일초 단위

의 정밀도를 가지는 INTEGER(8) 값을 출력한다.

irtc()함수의 사용 예제(Fortran)와 실행결과

```
   PROGRAM IRTC_TIME
   INTEGER SIZE
   PARAMETER(SIZE=5000)
   REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
   INTEGER(8) A, B, irtc
   A = irtc()
   DO i = 1, SIZE
      DO j = 1, SIZE
         N(i,j) = N(i,j) + M(i,j)*L(i,j)
      ENDDO
   END DO
   B = irtc()
   PRINT *, 'Nanoseconds elapsed: ',B - A
   END
```

```
$ irtc_time
 Nanoseconds elapsed : 19747222262
```

3.6.3 dtime_()

dtime_() 함수는 이전에 dtime_()이 호출된 시점으로부터 경과된 유저 CPU time과 시스템 CPU time을 백분의 일초 단위 정밀도로 출력한다.

dtime_()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM DTIME_TIME
REAL(4) DELTA, dtime_
TYPE CT_TYPE
SEQUENCE
   REAL(4) USRTIME
   REAL(4) SYSTIME
END TYPE
TYPE (CT_TYPE) DTIME_STRUCT

INTEGER ARRAY_SIZE
PARAMETER(ARRAY_SIZE=2000)

REAL*8 a(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 b(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 c(ARRAY_SIZE,ARRAY_SIZE)

DELTA = dtime_(DTIME_STRUCT)
DO i=1,2000
   DO j=1,2000
      c(i,j) = c(i,j) + a(i,j) * b(i,j)
   ENDDO
ENDDO

DELTA = dtime_(DTIME_STRUCT)
PRINT *, 'User time: ',DTIME_STRUCT%USRTIME, 'seconds'
PRINT *, 'System time: ',DTIME_STRUCT%SYSTIME, 'seconds'
PRINT *, 'Elapsed time: ',DELTA, 'seconds'
END
```

```
$ dtime_time
  User time: 1.570000052 seconds
  System time: 0.1599999964 seconds
  Elapsed time: 1.730000019 seconds
```

3.6.4 etime_()

etime_() 함수는 프로세스가 시작된 시점으로부터 경과된 유저 CPU time과 시스템 CPU time을 백분의 일초 단위 정밀도로 출력한다

etime_()함수의 사용 예제(Fortran)와 실행결과

```
   PROGRAM ETIME_TIME
   REAL(4) ELAPSED, etime_
   TYPE CT_TYPE
   SEQUENCE
      REAL(4) USRTIME
      REAL(4) SYSTIME
   END TYPE
   TYPE (CT_TYPE) ETIME_STRUCT

   INTEGER ARRAY_SIZE
   PARAMETER(ARRAY_SIZE=2000)

   REAL*8 a(ARRAY_SIZE,ARRAY_SIZE)
   REAL*8 b(ARRAY_SIZE,ARRAY_SIZE)
   REAL*8 c(ARRAY_SIZE,ARRAY_SIZE)

   DO i=1,2000
      DO j=1,2000
         c(i,j) = c(i,j) + a(i,j) * b(i,j)
      ENDDO
   ENDDO

   ELAPSED = etime_(ETIME_STRUCT)
   PRINT *, 'User time: ',ETIME_STRUCT%USRTIME, 'seconds'
   PRINT *, 'System time: ',ETIME_STRUCT%SYSTIME, 'seconds'
   PRINT *, 'Elapsed time: ', ELAPSED, 'seconds'
   END
```

```
$ etime_time
  User time: 1.789999962 seconds
  System time: 0.1800000072 seconds
  Elapsed time: 1.970000029 seconds
```

3.6.5 mclock()

함수 mclock()은 현재 프로세스의 유저 time과 모든 자식 프로세스의 유저 time과 시스템 time의 합을 출력한다. 출력 값의 단위는 100분의 일초 이다.

mclock()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM MCLOCK_TIME
INTEGER T1, T2, mclock
PARAMETER(SIZE=5000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
REAL seconds
T1 = mclock()
DO i = 1, SIZE
   DO j = 1, SIZE
      N(i,j) = N(i,j) + M(i,j)*L(i,j)
   ENDDO
END DO
T2 = mclock()
seconds = real(T2-T1)/100
PRINT *, ' Elapsed CPU time : ', seconds, 'seconds.'
END
```

```
$ mclock_time
 Elapsed CPU time: 18.75000000 seconds.
```

3.6.6 timef()

timef() 함수는 처음 timef가 호출된 순간을 0로 하여 그 시간부터 경과된 시간을 1000분의 일초 단위로 출력한다.

timef()함수의 사용 예제(Fortran)와 실행결과

```
   PROGRAM TIMEF_TIME
   INTEGER SIZE
   PARAMETER(SIZE=5000)
   REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
   REAL(8) elapsed, timef
   elapsed = timef()
   PRINT *, '  'Start time : ', elapsed
   DO i = 1, SIZE
      DO j = 1, SIZE
         N(i,j) = N(i,j) + M(i,j)*L(i,j)
      ENDDO
   END DO
   elapsed = timef()
   PRINT *, ' Elapsed time : ', elpased, 'milliseconds.'
   END
```

```
$ timef_time
  Start time : 0.00000000000000000E+00
  Elapsed CPU time: 21120.7036972045898
```

4.HPM Toolkit 사용법

HPM(Hardware Performance Monitor) Toolkit은 물리적인 프로세서내의 하드웨어 이벤트 카운터에서 사용자가 필요한 정보를 얻을 수 있도록 하는 라이브러리와 유틸리티의 모음으로 IBM에서 제공하고 있으며 현재 KISTI의 IBM 시스템에는 버전 2.4.3이 설치되어있다. HPM Toolkit은 hpmcount, libhpm, hpmviz 등으로 구성되어 있으며 다음과 같은 다양한 하드웨어 이벤트들을 측정할 수 있다.

① clock cycles
② instructions completed
③ L1 load/store misses
④ L2 load/store misses
⑤ TLB misses
⑥ FPU/FXU activity
⑦ number of branches
⑧ branch mispredictions
⑨ loads/stores completed
⑩ FMAs executed

POWER4 프로세서에서는 POWER3 프로세서에서와 마찬가지로 하드웨어 이벤트 카운터에서 동시에 8가지의 정보를 측정할 수 있다. IBM에서는 사용자가 사용하기에 편리하도록 하드웨어 이벤트 카운터 정보를 8개씩 묶어 0부터 60까지 총 61개의 그룹을 제공하고 있으며(POWER3의 경우는 4개의 그룹 제공) 디폴트로 사용되는 그룹은 60번이다. 각 그룹에서 다루는 정보들에 대해서는 IBM AIX 시스템의 /usr/pmapi/lib/POWER4.gps에 그 내용이 있다. 사용자가 프로그램의 성능분석에 자주 이용하는 정보를 담은 몇 개의 그룹들을 소개하면 다음과 같다.

① 그룹 60 : cycles, instructions, FP 연산(나누기, FMA, 로드, 저장 포함) 회수 등
② 그룹 59 : cycles, instructions, TLB 미스, 로드, 저장, L1 미스

회수 등
③ 그룹 5 : L2, L3, 그리고 메모리로부터의 로드 회수.
④ 그룹 58 : cycles, instructions, L3로부터의 로드, 메모리부터의 로드 회수 등
⑤ 그룹 53 : cycles, instructions, 고정소수점 연산, FP 연산(나누기, SQRT, FMA, and FMOV or FEST 포함) 회수 등

4.1 hpmcount

hpmcount는 사용자가 작성한 프로그램의 실제 실행 시간과 하드웨어 카운터에 관련된 정보, 사용 자원 등의 전반적인 성능을 제공하는 커맨드라인 유틸리티이다.

Sequential programs on AIX:

```
$hpmcount      [-o <filename>] [-n] [-g <group>] <program>
```

Parallel programs (MPI) on AIX:

```
$poe hpmcount [-o <filename>] [-n] [-g <group>] <program>
```

-o <filename> : 출력화일 <filename>.<pid> 생성옵션. 병렬 프로그램에서는 프로세스마다 하나의 파일이 생성되며, 디폴트는 표준출력.
-n : 표준출력을 하지 않고 파일로만 출력. -o옵션과 같이 사용됨.
-g <group> : (POWER4 only) 0에서 60까지 그룹 지정 가능하며, 디폴트는 60.

hpmcount 사용 예제와 실행결과(-g 60은 없어도 됨)

```
$ hpmcount -o hpmtest -n -g 60 a.out
 Seconds elapsed:  20.1621870994567871
$ vi hpmtest_0000.384218

 hpmcount (V 2.4.3) summary

 Total execution time (wall clock time): 3.890695 seconds

 ########  Resource Usage Statistics  ########

 Total amount of time in user mode            : 3.800000 seconds
 Total amount of time in system mode          : 0.060000 seconds
 Maximum resident set size                    : 23564 Kbytes
 Average shared memory use in text segment    : 3088 Kbytes*sec
 Average unshared memory use in data segment  : 9007560 Kbytes*sec
 Number of page faults without I/O activity   : 5893
 Number of page faults with I/O activity      : 0
 Number of times process was swapped out      : 0
 Number of times file system performed INPUT  : 0
 Number of times file system performed OUTPUT : 0
 Number of IPC messages sent                  : 0
 Number of IPC messages received              : 0
 Number of signals delivered                  : 0
 Number of voluntary context switches         : 4
 Number of involuntary context switches       : 3

 #######  End of Resource Statistics  ########

  PM_FPU_FDIV  (FPU  executed  FDIV  instruction)                          :
0
  PM_FPU_FMA  (FPU  executed  multiply-add  instruction)                   :
1000002329
  PM_FPU0_FIN  (FPU0  produced  a  result)                                 :
627844244
  PM_FPU1_FIN  (FPU1  produced  a  result)                                 :
1377820044
  PM_CYC  (Processor  cycles)                                              :
3758316603
  PM_FPU_STF  (FPU  executed  store  instruction)                          :
1003408033
  PM_INST_CMPL  (Instructions  completed)                                  :
4143170873
  PM_LSU_LDF   (LSU   executed   Floating   Point   load   instruction)   :
2002211584
```

```
Utilization rate                              :         74.306 %
                                              ①
Load and store operations                     :       3005.620 M
                                              ②
Instructions per load/store                   :          1.3786
                                              ③
MIPS                                          :       1064.892
                                              ④
Instructions per cycle                        :          1.102
                                              ⑤
HW Float points instructions per Cycle        :          0.534
                                              ⑥
Floating point instructions + FMAs            :       2002.259 M
                                              ⑦
Float point instructions + FMA rate           :     514.627 Mflip/s
                                              ⑧
FMA percentage                                :         99.887 %
                                              ⑨
Computation intensity                         :          0.666
                                              ⑩
```

위의 실행결과에서 수집된 정보에 대해 살펴보면 다음과 같다.

① Utilization rate = User time / Wall clock time
② Load and store operations = Loads + Stores(Total LS)
    2002211584 + 1003408033 = 3005619617 = 3005.620 M
③ Instructions per load/store = Instructions completed / Total LS
    4143170873 / 3005619617 = 1.3786
④ MIPS = 0.000001 * Instructions completed / Wall clock time
    0.000001 * 4143170873 / 3.890695 = 1064.892
    ※ MIPS(Millions of Instructions per second)
⑤ Instructions per cycle = Instructions completed / Cycles
    4143170873 / 3758316603 = 1.102
⑥ HW Float points instructions per Cycle = (FPU 0 + FPU 1) / Cycles
    (627844244 + 1377820044) / 3758316603 = 0.534
    ※ FPU(Floating-point Processing Unit : 부동 소수점 연산장치) POWER4에는 프로세서 하나에 2개씩 있음)
⑦ Floating point instructions + FMAs (flip) = FPU 0 instructions + FPU 1 instructions + FMAs executed - FPU Stores (POWER4)

627844244 + 1377820044 + 1000002329 - 1003408033 = 2002258584 = 2002.259 M

※ FMA (Floating-point Multiply/Add), FDIV(Floating-point Divide)

※ POWER3 : flip = FPU 0 instructions + FPU 1 instructions + FMAs executed

⑧ Float point instructions + FMA rate = 0.000001 * flip / Wall clock time (Mflip/s)

0.000001 * 2002258584 / 3.890695 = 514.627 Mflip/s

⑨ FMA percentage = 100 * FMAs executed * 2 / flip

100 * 1000002329 * 2 / 2002258584 = 99.887 %

⑩ Computation intensity = flip / Total LS

2002258584 / 3005619617 = 0.666

hpmcount를 이용해 얻은 이와 같은 많은 정보들 중에서 사용자가 관심을 둘 항목은 코드에 의해 수행된 부동소수점 연산의 효율을 나타내는 Mflip/s(Millions of FLoat Instructions Per Second) 이다. 이 값은 초당 수행된 부동 소수점 연산의 회수를 나타내며 Mflops와 동일한 정보를 준다. 위의 결과 514.627 (Mflip/s)가 IBM 1.0GHz POWER4 프로세서에서 실행시킨 것이라면 프로세서 하나의 이론 최고 성능(Peak Performance)이 4000Mflops 이므로 이론 최고 성능의 약 13%에 해당되는 결과가 되는 셈이다. 따라서 예제로 사용된 코드는 많은 부분을 최적화 시킬 필요가 있음을 알 수 있다.

4.2 libhpm

libhpm은 사용자 코드에서 호출하여 사용하는 Fortran/C/C++ 루틴들로 구성되는 라이브러리 이다. hpmcount가 코드 전체의 성능을 알아보는데 사용되는것에 반해 libhpm는 코드내에서 호출하여 특정 부분에 대한 성능을 알아보는 목적으로 사용된다. libhpm 라이브러리는 OpenMP 또는 스레드 프로그램을 지원한다. libhpm 루틴을 호출하여 사용하는 프로그램을 컴파일하는 경우 사용자는 적절한 라이브러리를 링크 시켜줘야 하는데, OpenMP 또는 스레드 프로그램의 경우는 libhpm_r을 링크 시켜야 한다. libhpm은 프로그램의 실행 중에 필요한 정보를 수집하게 되므로 루프 내부에서 호출하

여 사용하는 경우 프로그램 실행에 많은 부하를 줄 수도 있다.

4.2.1 주요 함수들

C/C++ : hpmInit(taskID, progName)
Fortran : f_hpminit(taskID, progName)
- taskID : 노드 ID를 나타내는 정수
- progName : 프로그램 이름을 나타내는 스트링

C/C++ : hpmStart(instID, label)
Fortran : f_hpmstart(instID, label)
- instID : 성능측정을 원하는 구간을 나타내는 정수, 100까지의 자연수 사용.
- label : hpmviz를 사용할 때 나타나는 스트링

C/C++ : hpmStop(instID)
Fortran : f_hpmstop(instID)
- 각 instID에 대해 hpmStart에 대응하는 hpmStop이 반드시 필요

C/C++ : hpmTstart(instID, label)
Fortran : f_hpmtstart(instID, label)
- 스레드를 사용하는 프로그램에서 사용

C/C++ : hpmTstop(instID)
Fortran : f_hpmtstop(instID)
- 스레드를 사용하는 프로그램에서 사용

C/C++ : hpmGetTimeAndCounters(numCounters, time, values)
Fortran : f_GetTimeAndCounters (numCounters, time, values)

- 함수가 호출될 때마다 hpmInit이 호출된 이후에 축적된 회수와 초 단위의 시간을 출력한다.
- numCounters : 접근 회수를 나타내는 정수
- time : 배정도 실수
- values : numCounters의 크기를 가지는 long long 배열

C/C++ : hpmGetCounters(values)
Fortran : f_GetCounters (values)
- hpmGetTimeAndCounters와 유사하며 단지 축적된 회수만 출력하여 부하를 최소화 한다.

C/C++ : hpmTerminate(taskID)
Fortran : f_hpmterminate(taskID)
- 출력파일을 생성한다. 만약 호출하지 않으면 아무런 정보도 얻을 수 없다.

4.2.2 출력

각 프로세스 마다 다음과 같은 두개의 파일을 생성한다.
- perfhpm[taskID].[pid] : 성능 관련 데이터를 포함하는 텍스트 파일
- hpm[taskID]_[progName]_[pid].viz : hpmviz 프로그램을 위한 데이터를 포함하는 파일. 필요 없으면 환경변수 HPM_VIZ_OUTPUT = FALSE로 설정.

이때 생성되는 파일에 수집되는 정보는 hpmcount에서 사용된 하드웨어 이벤트 카운터 집합에 포함되는 정보와 동일하다. 즉, POWER4 시스템의 경우 하드웨어 이벤트 정보를 분류한 61개(0 ~ 60)의 그룹 중의 하나를 지정해 원하는 정보를 출력하도록 할 수 있다. 그룹의 선택은 환경변수 HPM_EVENT_SET을 이용해 지정하며 디폴트는 60이다.

4.2.3 컴파일과 링크

libhpm 라이브러리를 사용하기 위해서 프로그램 작성시 C/C++는 “libhpm.h”를 Fortran에서는 “f_hpm.h”를 각각 Include 시켜야 한다. 작성된 프로그램을 링크하는 과정에서는 libpmapi.a, libhpm.a(또는 libhpm_r.a), 그리고 liblm.a를 링크시켜야 한다. 확장자가 f인 Fortran 프로그램의 경우는 컴파일할 때 -qsuffix=cpp=f 옵션을 주어야 한다. 아래 예제의 컴파일 과정에서는 시스템에 HPM 사용을 위한 경로(path) 지정이 되어있지 않아 -I(include 경로 지정) -L(라이브러리 경로 지정) 옵션을 주고 현재 KISTI IBM 시스템에 설치된 HPM의 경로를 모두 지정해 주었다.

4.2.4 사용 예

libhpm 사용 예제(Fortran) 프로그램 : hpmtest.f

```
PROGRAM HPMTEST
#include “f_hpm.h”
INTEGER SIZE
PARAMETER(SIZE=2000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
INTEGER taskID
taskID = 0
CALL f_hpminit( taskID, “hpmtest” )
CALL f_hpmstart( 1, “Do Loop” )
DO i = 1, SIZE
   DO j = 1, SIZE
      N(i,j) = N(i,j) + M(i,j)*L(i,j)
   ENDDO
END DO
CALL f_hpmstop( 1 )
CALL f_hpmterminate( taskID )
END
```

컴파일과 링크

```
$xlf -o hpmtest hpmtest.f -I/applic/hpm_2_4_3/include -L/applic/hpm_2_4_3/lib
-lhpm -lpmapi -lm -qsuffix=cpp=f
```

생성된 성능정보 저장 텍스트 파일 : perfhpm0000.2744552

```
$ vi perfhpm0000.2744552

libhpm (Version 2.4.3) summary - running on POWER4

 Total execution time of instrumented code (wall time): 2.027903 seconds

 ########  Resource Usage Statistics  ########

 Total amount of time in user mode            : 1.850000 seconds
 Total amount of time in system mode          : 0.220000 seconds
 Maximum resident set size                    : 94708 Kbytes
 Average shared memory use in text segment    : 25172 Kbytes*sec
 Average unshared memory use in data segment  : 13301940 Kbytes*sec
 Number of page faults without I/O activity   : 23677
 Number of page faults with I/O activity      : 0
 Number of times process was swapped out      : 0
 Number of times file system performed INPUT  : 0
 Number of times file system performed OUTPUT : 0
 Number of IPC messages sent                  : 0
 Number of IPC messages received              : 0
 Number of signals delivered                  : 0
 Number of voluntary context switches         : 22
 Number of involuntary context switches       : 10

 #######  End of Resource Statistics  ########

 Instrumented section: 1 - Label: Do Loop - process: 0
 file: hpmtest.f, lines: 9 <--> 15
  Count: 1
  Wall Clock Time: 2.027757 seconds
  Total time in user mode: 1.37717590153846 seconds

  PM_FPU_FDIV (FPU executed FDIV instruction)              :            0
  PM_FPU_FMA (FPU executed multiply-add instruction)       :      4003433
  PM_FPU0_FIN (FPU0 produced a result)                     :      7989366
  PM_FPU1_FIN (FPU1 produced a result)                     :        24891
  PM_CYC (Processor cycles)                                :   1790328672
```

```
PM_FPU_STF (FPU executed store instruction)              :       4010925
PM_INST_CMPL (Instructions completed)                    :     112055809
PM_LSU_LDF (LSU executed Floating Point load instruction) :      12156326

Utilization rate                          :          67.916 %
Load and store operations                 :          16.167 M
Instructions per load/store               :           6.931
MIPS                                      :          55.261
Instructions per cycle                    :           0.063
HW Float points instructions per Cycle    :           0.004
Floating point instructions + FMAs        :           8.007 M
Float point instructions + FMA rate       :           3.949 Mflip/s
FMA percentage                            :         100.001 %
Computation intensity                     :           0.495
```

4.3 hpmviz

hpmviz는 libhpm라이브러리를 호출하여 생성한 성능분석결과를 GUI 기반으로 보여주는 유틸리티이다. 위의 예제(hpmtest.f)를 실행한 결과로 생성된 hpm0000_hpmtest_2744552.viz를 보기 위하여 적절한 X-윈도우 설정을 한 후 다음과 같이 실행 시킨다.

```
$ hpmviz hpm0000_hpmtest_2744552.viz
```

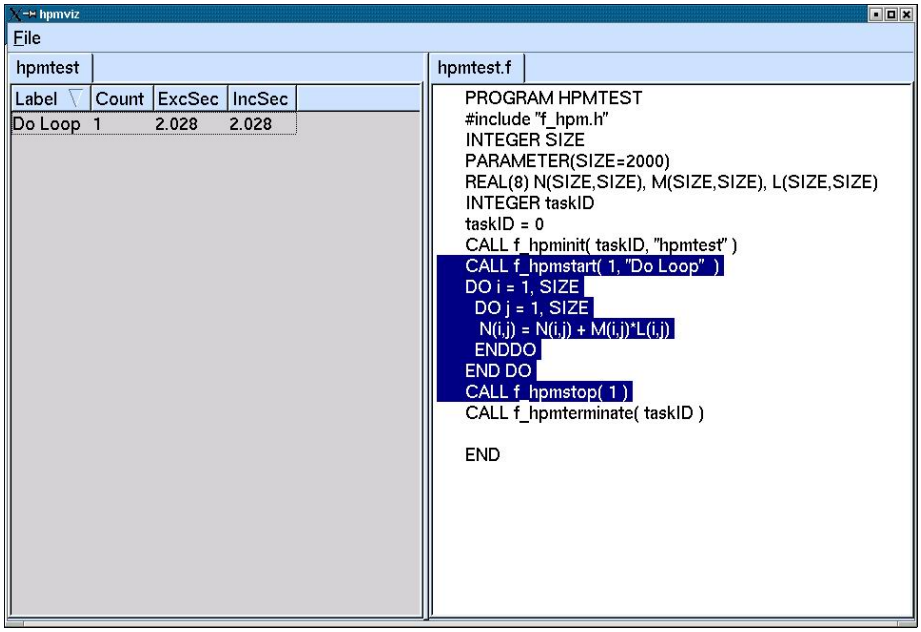


그림 4.1 실행결과

열려진 창의 좌측은 코드내에 성능 측정을 위해 hpmlib 루틴을 삽입한 부분과 그 부분의 실행시간 정보를 표시하고 우측 창에는 실제 코드를 표시한다. 좌측 창에서 hpmlib 루틴을 삽입한 부분을 선택하여 마우스 오른쪽 버튼을 누르면 그 부분에 대한 보다 자세한 HPM 정보를 보여주는 새로운 창이 하나 뜬다.

5. PCT와 Jumpshot을 이용한 MPI 프로그램 성능 분석

5.1 PE Benchmarker

PE Benchmarker는 IBM AIX Parallel Environment 환경에서 구동 되는 프로그램들의 성능을 분석할 수 있는 응용프로그램과 도구들로 구성된 툴셋이며, 크게 다음 세가지로 이루어져 있다.

- PCT(Performance Collection Tool) : 하나 혹은 그 이상의 응용프로그램 프로세스로부터 MPI 이벤트 트레이스 데이터 혹은 하드웨어/운영체제 성능 데이터를 수집한다. 커맨드라인 인터페이스 모드와 그래픽 사용자 인터페이스(GUI) 모드로 실행 가능하다.
- UTE(Unified Trace Environment) 도구 모음 : PCT를 이용해 수집된 MPI 이벤트 트레이스 정보를 저장해둔 AIX 트레이스 파일들을 UTE interval 파일로 변환시키고, 변환된 파일로부터 성능분석 테이블을 생성한다. 다음과 같은 UTE 도구들이 있으며 커맨드라인 인터페이스 모드로 실행된다.
  - uteconvert : AIX 이벤트 트레이스 파일을 UTE interval 파일로 변환한다.
  - utemerge : 여러 개의 UTE interval 파일을 하나의 interval 파일로 합친다.
  - utestats : UTE interval 파일로부터 얻은 정보에 대한 통계 테이블을 생성한다.

- slogmerge : Argonne 국립 연구소의 MPI 프로그램 성능분석 도구인 Jumpshot을 이용할 수 있도록 UTE interval 파일들을 SLOG(Scalable logfile) 파일 포맷으로 변환시키고 합친다.

- PVT(Performance Visualization Tool) : PCT를 이용해 하드웨어/운영체제 성능 데이터를 수집하면, 각 프로세스별로 수집된 프로파일 정보는 netCDF(network Common Data Form) 파일로 저장된다. PVT는 netCDF 파일로부터 프로파일 정보를 읽고 요약해 사용자에게 보여준다. 커맨드라인 모드와 GUI 모드로 실행 가능하다.

PE Benchmarker 툴셋을 이용한 프로그램의 성능분석은 PCT와 PVT를 이용한 하드웨어/운영체제 성능분석과 PCT와 UTE 혹은 PCT와 UTE 그리고, Jumpshot을 이용한 MPI 프로그램 성능분석으로 나누어 볼 수 있다.(그림 1) 이중 PCT와 PVT를 이용한 하드웨어/운영체제 성능분석은 2003년 1사분기 HPC 기술서에 소개된 HPM 툴킷과 그 기능이 유사하다. 이번 HPC 기술서에서는 PCT와 UTE를 이용하여 MPI 프로그램의 성능 정보를 수집하고 Argonne 연구소의 Jumpshot을 이용하여 수집된 정보를 GUI 환경에서 분석하는 방법을 소개한다.

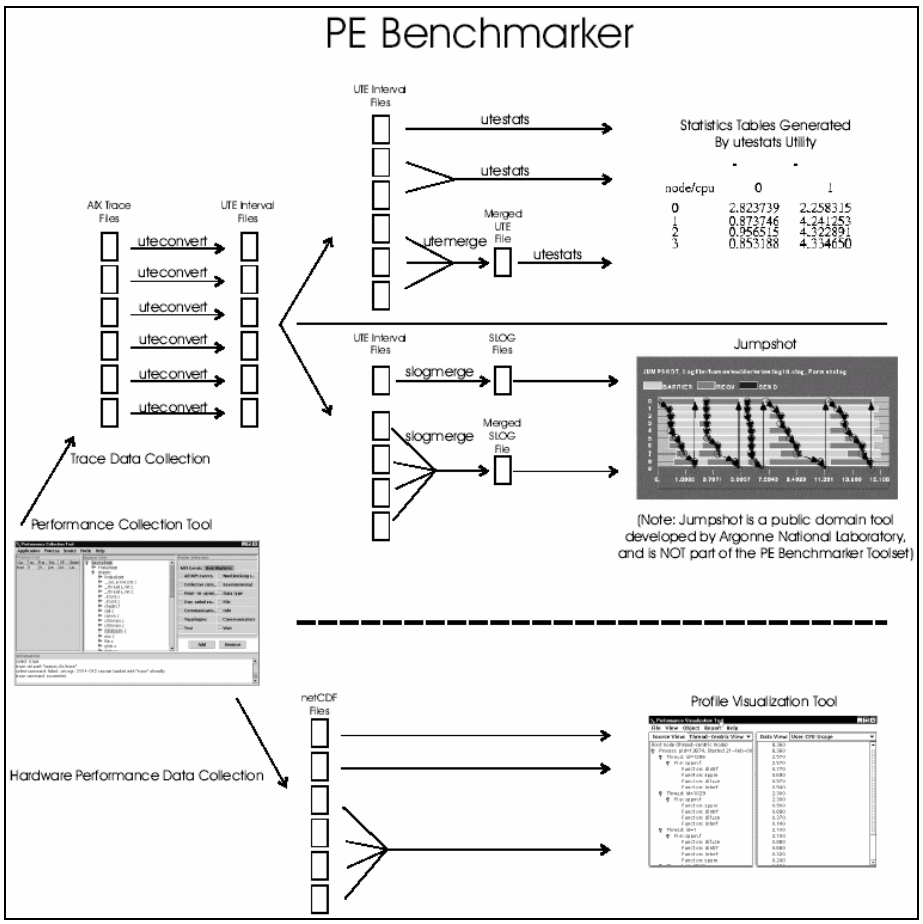


그림 5.1 PE Benchmarker를 이용한 프로그램 성능분석

5.2 MPI 프로그램 프로파일을 위한 PCT 사용법

앞서 설명했듯이 PCT는 커맨드라인 인터페이스와 GUI를 모두 지원한다. 여기서는 GUI를 기준으로 하여 PCT의 사용법을 설명할 것이다. 커맨드라인 인터페이스에서 PCT를 사용하는 것에 대해서는 IBM의 PE for AIX V3R2.0

Operation and Use, Vol. 2의 105 페이지에 소개되어 있다.

Jumpshot을 이용한 MPI 프로그램 성능분석을 위하여, 분석에 사용될 MPI 트레이스 파일을 PCT를 이용하여 생성해야 한다. 이를 위해 성능분석이 필요한 프로그램을 컴파일하기 전에 먼저 필요한 라이브러리를 링크시켜야 하는데, 이는 환경변수 MP_UTE를 yes로 설정해두면 된다. 만약 사용자의 UNIX 쉘 환경이 ksh이면 export MP_UTE=yes 명령어를 커맨드라인 상에서 입력하면 된다. 그리고, 이렇게 설정된 MP_UTE 환경변수를 통해 UTE 라이브러리를 추가할 수 있도록 하기위해서는 컴파일 스크립트를 반드시 "_r" 버전을 사용해야 한다.

1) 환경변수 설정

```
$ export MP_UTE=yes
```

2) 프로그램 컴파일

성능분석을 위한 MPI 프로그램을 스레드 사용 가능한(thread-enabled) 컴파일 스크립트(_r)를 이용하여 컴파일 한다. 다음에 사용된 pipelined.f 코드는 2차원 Laplace 방정식을 푸는 프로그램으로 IBM 레드북 PE for AIX V3R2.0, Hitchhiker's Guide에 소개되어 있다.

```
$ mpxlf_r -o pipelined pipelined.f
```

3) PCT 실행

사용자의 시스템에서 PCT를 GUI로 사용하기 위해서는 적절한 X윈도우 환경이 우선적으로 설정되어 있어야 한다.

```
$ pct
```

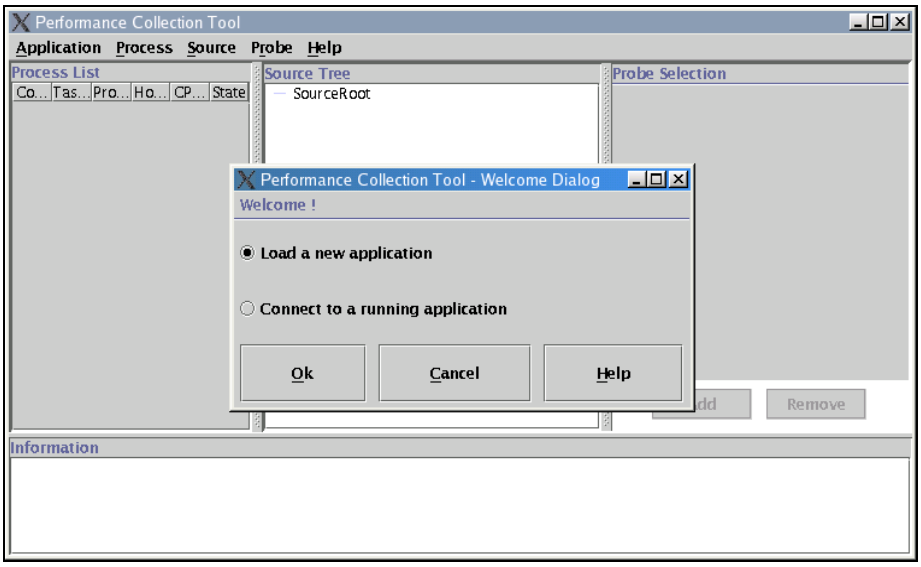


그림 5.2 PCT 실행 화면

① Load a new application을 선택하고 OK를 클릭한다. (Load Application 윈도우가 열린다.)

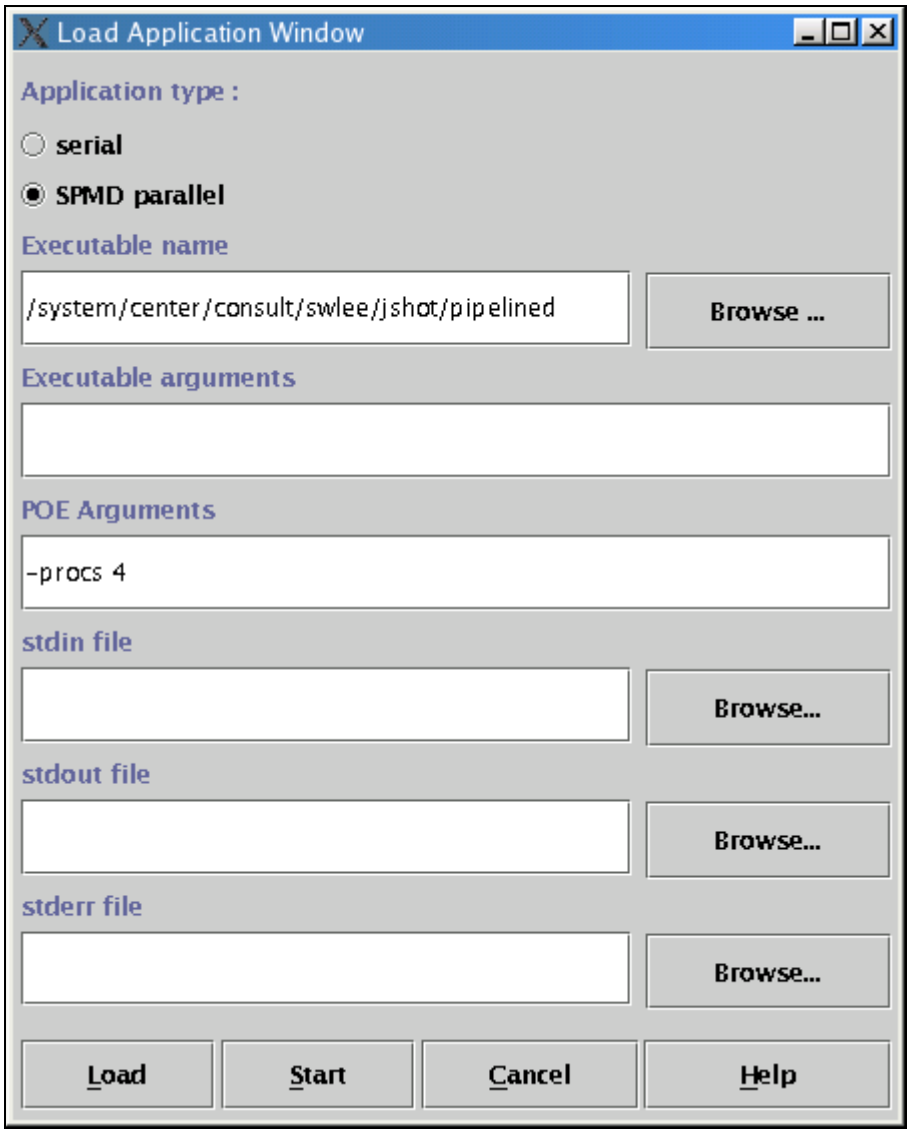


그림 5.3 Load Application 윈도우

② Load Application 윈도우에서 SPMD parallel을 선택하고 Executable name 필드에서 Browse 버튼을 이용하여 성능분석이 필요한 프로그램을 선택한다.

③ POE Arguments 필드에 적절한 POE 실행 옵션을 넣는다. 예를 들면, 병렬실행에 참여하는 프로세스 개수(-procs n) 등을 넣어준다.

④ Load 버튼을 클릭하여 프로그램을 로드한다. (Probe Data Selection 윈도우가 열린다.)

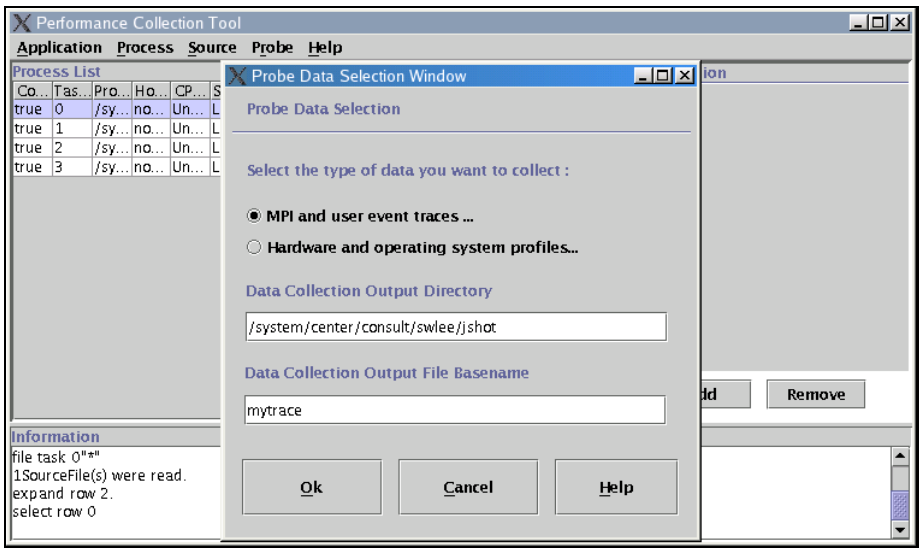


그림 5.4 Probe Data Selection 윈도우

⑤ 수집을 원하는 데이터의 타입을 선택하는데 지금과 같이 MPI 프로그램 성능분석을 위해서는 MPI and user event traces를 선택해야 하지만 만약 하드웨어/운영체제 성능 데이터를 원한다면 Hardware and operating system profiles를 선택해야 한다.

⑥ 출력파일이 생성될 디렉터리와 기본파일명을 적어주고 OK를 클릭한다. (메인 윈도우가 열린다.)

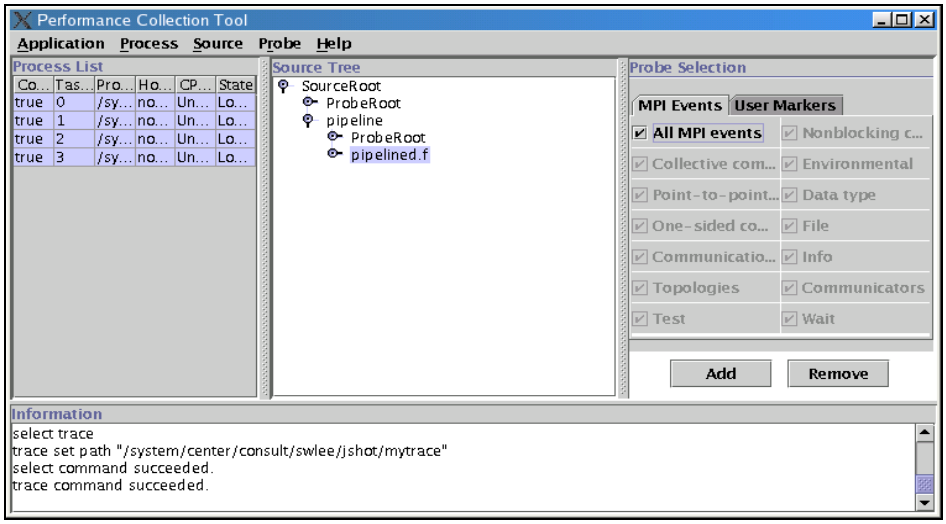


그림 5.5 PCT 메인 윈도우

⑦ 모든 프로세스의 진행 상황을 보기위해 메인 윈도우에서 Process → Select All Connected 를 선택한다.

⑧ Source Tree에서 pipelined.f를 선택한다.

⑨ 트레이스 정보를 수집하기 위해 Probe Selection 영역에서 All MPI Events를 선택한다.

⑩ Add 버튼을 클릭해 필요한 정보 수집을 위한 탐침(probe)을 설치한다.

⑪ 메뉴 바에서 Application → Start를 선택해 프로그램을 실행시킨다.

⑫ 프로그램의 실행이 완료되면 Target Application Exited 윈도우가 나타나고 OK버튼을 클릭하면 PCT가 종료된다.

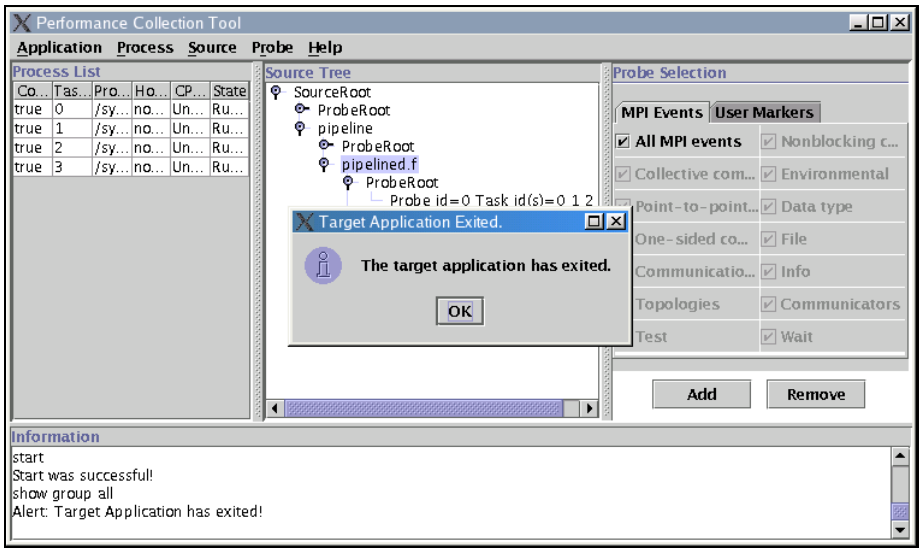


그림 5.6 PCT를 이용한 정보수집 완료

위와 같은 과정을 거쳐 성공적으로 메시지 패싱에 대한 데이터를 수집했다면 AIX 트레이스 파일이 생성된다. 이 트레이스 파일은 각 노드마다 하나씩 생성되는데, 위의 예에서 사용한 네 개의 프로세스 모두를 하나의 노드에서 실행되도록 했다면 트레이스 파일은 mytrace.#의 형식으로 하나가 생성된다. 여기서 #은 프로세스가 네 개이므로 0부터 3까지의 값 중 임의의 값을 하나 가지게 된다.

4) uteconvert 실행

생성된 AIX 트레이스 파일을 uteconvert를 이용하여 UTE interval 파일로 변환한다. 기본적으로 트레이스 파일의 이름을 그대로 가져와 mytrace.ute.# 파일이 생성되지만 -o 옵션을 이용하여 원하는 이름을 지정해 줄 수 있다.

```
$ uteconvert mytrace.1
```

```
$ uteconvert -o tracefile.ute mytrace.1
```

5) slogmerge 실행

수집한 정보를 Jumpshot에서 볼 수 있도록 slogmerge를 이용해 UTE interval 파일을 SLOG 파일로 변환한다. 기본적으로 (UTE 파일 이름).slog 파일이 생성되지만 역시 -o 옵션을 이용하면 원하는 이름을 지정해 줄 수 있다.

```
$ slogmerge mytrace.ute.1
```

```
$ slogmerge -o tracefile.slog tracefile.ute
```

5.3 Jumpshot 사용법

Jumpshot은 PE Benchmarker 툴셋에 포함된 프로그램이 아니며 이는 미국 Argonne 국립 연구소에서 개발한 공개 MPI 구현 패키지인 MPICH/MPE에 포함된 자바기반의 성능분석 가시화 툴이다. 이는 현재 KISTI IBM 시스템의 /applic/bin 디렉토리에 jumbshot이라는 이름으로 설치되어 있으며 인터넷에서 받아 개별적으로 설치해 사용할 수도 있다. GUI 기반의 Jumpshot도 PCT와 마찬가지로 실행하기 전에 적절한 X윈도우 설정이 필요하다.

성능분석을 원하는 MPI 프로그램의 트레이스 파일 생성부터 이것을 다시 SLOG 파일 포맷으로 변환시키는 과정까지 무사히 마쳤다면 이제 Jumpshot을 실행해 트레이스 파일 정보를 가시화시켜 보자.

1) Jumpshot 실행

```
$ /applic/bin/jumpshot
```

① 메뉴 바에서 File → Select Logfile을 이용해 만들어둔 SLOG 파일을 로드한다.(View and Frame Selector 윈도우가 열린다.)

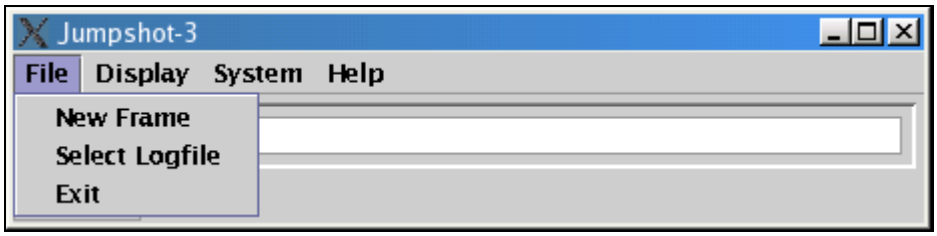


그림 5.7 Jumpshot 실행화면

② View and Frame Selector 윈도우에는 Event Count vs. Time의 그래프가 나타난다. 프로그램의 진행시간에 따른 적절한 프레임을 선택하고 View Options에서 MPI-Process를 선택한 후 Display 버튼을 클릭하면 Time line 윈도우를 볼 수 있다.

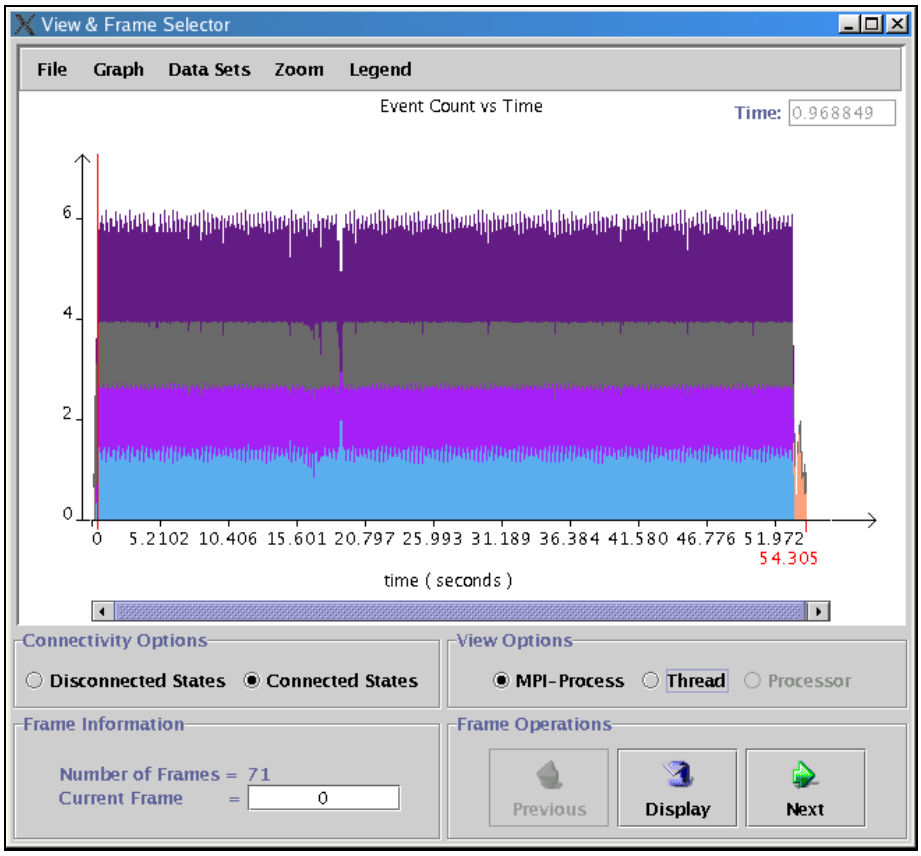


그림 5.8 View and Frame Selector 윈도우

③ Time Line 윈도우의 X축은 프로그램의 진행시간을 Y축은 프로세스 랭크를 나타낸다. MPI 서브루틴은 하나의 box로 표현되고 함수 내부 혹은 함수 사이의 통신은 화살표로 표시된다. 상단의 Zoom Operations에서 In/Out 버튼을 이용하여 선택한 프레임의 MPI 이벤트 발생을 좀더 자세히 살펴볼 수 있다.

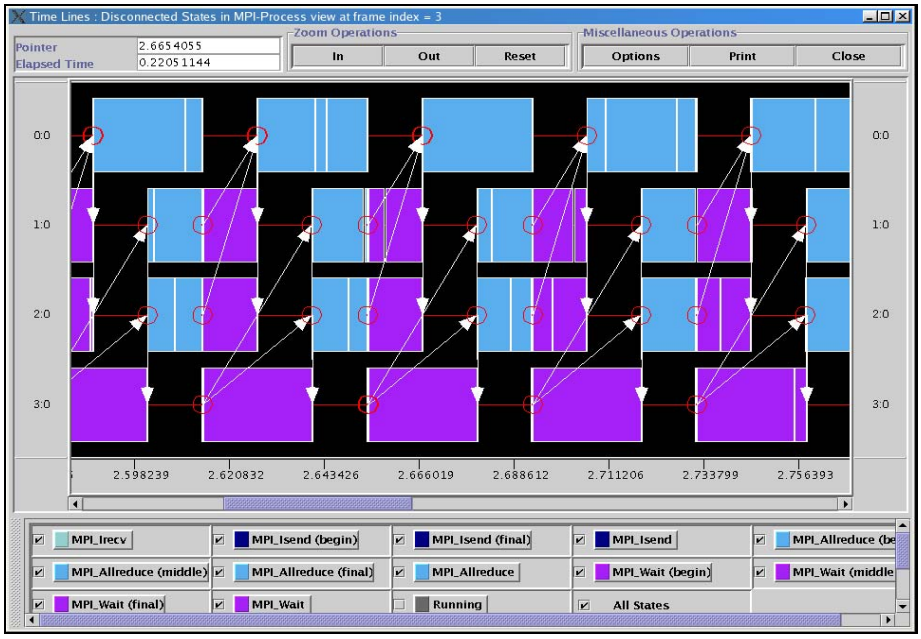


그림 5.9 Time Line 윈도우

사용자는 Time Line 윈도우의 MPI 프로그램 진행상황과 통신특성 등을 살펴보고 자신의 프로그램에 대한 성능분석을 한 후 적절한 최적화 방안을 마련할 수 있다. 가령 위의 그림을 예로 보면 프로세스들이 작업을 진행하기 전에 다른 프로세스들의 작업완료를 기다리며 많은 시간을 낭비하고 있음을 볼 수 있다. 즉 프로세스 랭크 1, 2, 3의 실행에서 밝은 보라색으로 표시된 부분이 MPI_Wait의 실행을 하면서 대기하는 것을 나타내고 있는데, 특히 3번 프로세스의 경우 대기시간이 대부분을 차지하고 있음을 확인할 수 있다.

5.4 관련정보

이상에서 PCT와 Jumpshot을 어떻게 실행하고 또, 어떻게 MPI 병렬 프로

그램의 프로파일 정보를 얻을 수 있는지 간략히 알아 보았다.

현재 SLOG 파일 포맷의 기능을 보다 향상시킨 SLOG-2 파일 포맷 개발 프로젝트가 진행 중이며 초기 구현 버전이 발표되어 있다. SLOG-2 파일 포맷을 지원하는 Jumpshot-4(앞서 다루었던 내용은 Jumpshot-3를 기준으로 한 것임)도 현재 발표 되어 있으며, 이와 관련된 내용은 아래의 Argonne 국립 연구소 홈페이지를 참고하기 바란다.

- PCT 관련정보
- IBM Redbook: PE for AIX V3R2.0: Hitchhiker's Guide
- IBM Redbook: PE for AIX V3R2.0 Operation and Use, Vol. 2

- SLOG와 Jumpshot 관련정보
- Argonne 연구소:
- http://www-unix.mcs.anl.gov/perfvis/software/index.htm

6. 모니터링 툴 사용법(ps, nmon)

6.1 ps 사용법

ps명령은 프로세서의 현재 상태를 표시한다. 즉 표준 출력에 활동 중이 프로세서의 현재 상태와 관련 커널 스레드를 표시한다. 다음과 같은 2개의 옵션과 플래그를 혼합하여 많이 사용된다.

```
(1) ps -ef[my_id@nobel my_id]$ ps -ef
     UID    PID   PPID   C    STIME    TTY  TIME CMD
    root      1      0   0   Jul 24      - 124:43 /etc/init
    root   5460   9304   0   Aug 02      -  0:00 dtlogin <:0>        -
daemon
    root   6318      1   0   Jul 24      - 321:28 /usr/sbin/syncd 60
    root      6614     19358      0      Jul  24             -    0:00
/usr/opt/ifor/bin/i4llmd -b -n wcclwts -l /var/ifor/llmlg
    root      6970           1      0      Jul  24           -    0:00
/usr/lib/methods/ssa_daemon -l ssa1
    root      7234           1      0      Jul  24           -    0:00
/usr/lib/methods/ssa_daemon -l ssa0
    root      7484           1      0      Jul  24           -    0:00
/usr/lib/methods/ssa_daemon -l ssa2
    root   8156   8844   0   Jul 24      -  0:00 /usr/sbin/writesrv
    root   8296   8844   0   Jul 24      -  0:53 /usr/sbin/inetd
    root      8516           1      0      Jul  24           -    0:00
/usr/lib/methods/ssa_daemon -l ssa3
    root   8844      1   0   Jul 24      -  0:00 /usr/sbin/srcmstr
    root   9304      1   0   Jul 24      -  5:23 /usr/dt/bin/dtlogin -
daemon
    root   9860   8844   0   Jul 24      -  0:06 /usr/sbin/syslogd
    root  10098      1   0   Jul 24      -  5:13 /usr/sbin/cron
    root  10366      1   0   Jul 24      -  0:00 /usr/ccs/bin/shlap
    root  10602   8844   0   Jul 24      - 292:38 /usr/sbin/portmap
    root  10866   9304   0   Jul 24      -  9:58 /usr/lpp/X11/bin/X -x
```

```
/dev/console
    root     16038       8844       0       Oct   20               -     1:00
/usr/sbin/rsct/bin/IBM.CSMAgentRMd
    root   16512      1   0   Jul 24      -  0:00 /usr/bin/AIXPowerMgtDaemon
    root     17802              1       0       Jul   24               -     0:00
/usr/lpp/diagnostics/bin/diagd


    root   18318   8844   0   Jul 24      -  9:20 /usr/sbin/rsct/bin/ctcasd
    root   19358   8844   0   Jul 24      - 34:55 /usr/opt/ifor/bin/i4llmd -
b -n wcclwts -l /var/ifor/llmlg
    root    26490     8844     0     Jul  24          -    0:01  java  -Xmx128m  -
DMessageLang=C          -Ddisplay=nobel:0.0          -DWINDOWID=           -
DWEBSM_ALL_PERMISSIONS_FOR_SECURE=true com.ibm
.websm.refresh.WSMRefreshServer
    root   29698   8296   0   Jul 24      -  0:22 rpc.ttdbserver 100083 1
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

ps에 -ef플래그를 붙여 사용함으로써 커널 프로세스를 제외한 모든 프로세스에 대한 전체 정보 리스트를 보여 준다. 여기서 열 방향 정보에 대한 헤드의 의미는 다음과 같다.

UID : 프로세스 소유자의 사용자 ID

PID : 프로세스의 프로세스 ID

PPID : 상위 프로세스의 프로세스 ID

C : 프로세스나 스레드의 CPU 사용량

STIME : 프로세스의 시작 시간

TTY : 프로세스를 위한 제어 웍스테이션

- 이 프로세스는 웍스테이션과 연관되어 있지 않다.

? 알려지지 않음

Number TTY 번호

TIME : 프로세서에 대한 총 수행 시간

CMD : 커맨더 이름을 나타낸다.

```
 [my_id@nobel my_id]$ ps aux
USER        PID %CPU %MEM    SZ  RSS    TTY STAT    STIME  TIME COMMAND
root        516 16.1  0.0    12 15892      - A      Jul 24 136812:10 wait
root        774 15.7  0.0    12 15892      - A      Jul 24 133703:08 wait
root       1290 15.7  0.0    12 15892      - A      Jul 24 133337:48 wait
root       1032 15.6  0.0    12 15892      - A      Jul 24 133178:12 wait
root       1806 15.6  0.0    12 15892      - A      Jul 24 132494:49 wait
root       1548 15.5  0.0    12 15892      - A      Jul 24 132084:19 wait
root      12650  0.3  0.0    20 15896      - A      Jul 24 2942:51 nfsd
root       260378   0.3   0.0  7448  3556        - A        Oct  17  301:56
LoadL_startd -f
root       6726  0.1  0.0    20 15900      - A      Jul 24 798:31 kbiod
root     159010  0.1  0.0 4976 1848      - A      Sep 24 158:39 nmon
root        239508   0.0   0.0  13124  8828         - A     Oct  17  42:59
                                                    LoadL_negotiat
                                                    or
root       6318  0.0  0.0  164  140      - A      Jul 24 321:37
/usr/sbin/syncd
root      10602  0.0  0.0 1968  708      - A      Jul 24 292:43
/usr/sbin/portma
root     198112  0.0  0.0 4528 1648      - A      Sep 11 124:27 nmon
root       3096  0.0  0.0   64 15936      - A      Jul 24 190:24 gil
root       2322  0.0  0.0   12 15888      - A      Jul 24 129:13 lrud
root          1  0.0  0.0 1868  176      - A      Jul 24 124:44 /etc/init
root          0  0.0  0.0   16 15896      - A      Jul 24 59:55 swapper
root     260046  0.0  0.0 6608 4368      - A      Oct 17  6:43 LoadL_schedd
-f
loadl    249336  0.0  0.0 4232 2344 pts/44 A     Oct 23  3:16 xloadl
root         19358   0.0   0.0  4212  1516        - A         Jul  24  35:02
/usr/opt/ifor/bi
root       9620  0.0  0.0   16 15892      - A      Jul 24 32:47 jfsz
root        139562   0.0   0.0   524   428        - A         Aug  01  12:38
/usr/lib/netsvc/
loadl     84638  0.0  0.0 3960 3948      - A    10:29:44  0:02 ftpd -u 022
root      13936  0.0  0.0   20 15896      - A     Jul 24 10:56 rpc.lockd
root       267198   0.0   0.0  4324  1132        - A         Oct  17   1:22
/usr/lpp/LoadL/f
root         10866   0.0   0.0  14328  2056        - A         Jul  24   9:58
/usr/lpp/X11/bin
root         16038   0.0   0.0  2772   592        - A         Oct  20   1:00
/usr/sbin/rsct/b
e940scs  200592  0.0  0.0 2320 2300      - A      Oct 28  0:12 ftpd -u 022
root         15232   0.0   0.0  3320  1052        - A         Jul  24   9:29
/usr/sbin/rsct/b
root         18318   0.0   0.0  2416   424        - A         Jul  24   9:20
```

```
/usr/sbin/rsct/b
root        164836   0.0   0.0  2604  2208         -  A         Oct  25   0:29
/usr/sbin/rsct/b
root        103140   0.0   0.0  2436  2268         -  A         Oct  29   0:07
/usr/sbin/rsct/b
root         91554   0.0   0.0  3260  3080         -  A         Oct  26   0:23
/usr/sbin/rsct/b
i002res  257288  0.0  0.0 2316  628      -  A      Oct 24  0:26 ftpd -u 022
root          9304   0.0   0.0  1652   524         -  A         Jul  24   5:23
/usr/dt/bin/dtlo
root         10098   0.0   0.0  2360   684         -  A         Jul  24   5:13
/usr/sbin/cron
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

ps 명령에 aux옵션을 붙여 모든 프로세스에 대한 정보를 사용자 위주로 출력할 수 있다. 이 명령의 열 방향 정보에 대한 헤드의 의미는 다음과 같다.

```
USER : 프로세스 소유자의 로그인 이름
PID : 프로세스의 프로세스 ID
%CPU : 프로세서 시작 이후에 프로세스가 CPU를 사용한 시간의 백분율, 이 값은
프로세스가 CPU를 사용한 시간을 프로세스의 경과시간으로 나누어 계산된다.
%MEM : 프로세스에 의해 사용된 실제 메모리 비율
SZ : 프로세스의 코어 이미지 크기(IKB 단위)
RSS : 프로세스이 실제 메모리 크기(1KB 단위)
TTY : 프로세스를 위한 제어 윅스테이션
        - 이 프로세스는 윅스테이션과 연관되어 있지 않다.
        ? 알려지지 않음
        Number TTY 번호
STAT : 다음과 같이 프로세스 상태를 표시
        0 : 존재하지 않음
        A : 사용 중
        I : 중간
        Z : 취소
        T : 중지
```

```
        K : 사용 가능한 커널 프로세스
STIME : 프로세스의 시작 시간
TIME : 프로세서에 대한 총 수행 시간
COMMAND : 커맨더 이름을 나타낸다.
```

일반적으로 ps명령은 그 자체보다는 grep 명령과 함께 사용자가 실행시킨 프로그램에 대한 정보를 확인하기 위해 많이 사용한다. 즉 다음 예와 같이 ps명령다음에 파이프(|)를 하고, 사용자 아이디를 grep의 플래그로 사용함으로써 그 사용자에 대한 프로세스 정보만 추출하여 확인해 볼 수 있다.

```
[my_id@nobel my_id]$ ps -ef|grep my_id
  my_id  61900  33154   0   Aug 13 pts/61  0:00 -ksh
  my_id  75282 187294   0 13:42:10 pts/12  0:00 -bash
  my_id 158514 234388   0 13:40:06  pts/6  0:00 -bash
  my_id 190644 269678   0 14:19:51 pts/38  0:00 -bash
  my_id 226304 190644  45 18:02:21 pts/38  0:00 ps -ef
  my_id 241608 190644   0 18:02:21 pts/38  0:00 grep my_id
  my_id 256078  75282   0 18:02:15 pts/12  0:00 ksh ./run.omp 2
  my_id 259504 256078 240 18:02:15 pts/12  0:10 pi_omp
```

```
[my_id@nobel my_id]$ ps aux|grep my_id
my_id   259504 31.9  0.0 1440 1444 pts/12 A    18:02:15  0:23 pi_omp
my_id   226306  0.0  0.0  616  632 pts/38 A    18:02:27  0:00 ps aux
my_id   256078  0.0  0.0  636  668 pts/12 A    18:02:15  0:00 ksh ./run.omp
2
my_id   241610  0.0  0.0  296  304 pts/38 A    18:02:27  0:00 grep my_id
my_id    61900  0.0  0.0 1076  716 pts/61 A      Aug 13  0:00 -ksh
my_id    75282  0.0  0.0 3388 3044 pts/12 A    13:42:10  0:00 -bash
my_id   158514  0.0  0.0 3380 3036  pts/6 A    13:40:06  0:00 -bash
my_id   190644  0.0  0.0 3388 3044 pts/38 A    14:19:51  0:00 -bash
```

6.2 nmon 사용법

nmon은 실시간으로 계속 시스템을 모니터링할 수 있는 아주 간단한 툴이다. 이 프로그램은 O/S의 기본 패키지에 포함이 된 것이 아니라 추가적으로 설

치를 해야 하는 것으로 현재 IBM IBM 시스템에는 전 노드에 설치가 되어 있다. 이 프로그램을 실행하기 위해서는 간단하게 command line에서 nmon 하고 실행하면 된다.

[my_id@nobel1a my_id]$ nmon

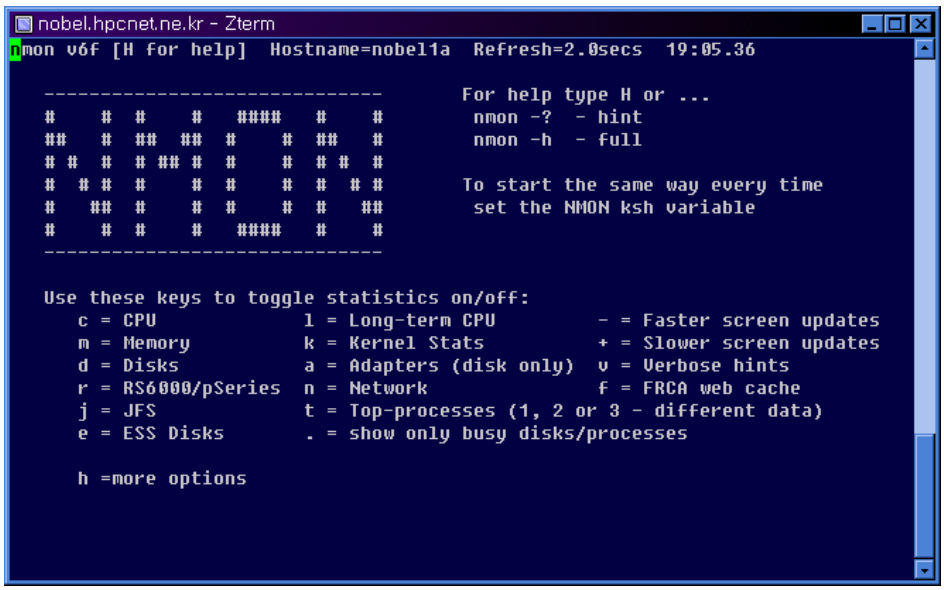


그림 6.1 nmon 실행화면

그러면 위와 같은 간단한 사용법이 포함된 화면이 출력된다. 이 상태에서 아래부분에 간략하게 설명된 키를 누르면 그에 해당하는 정보를 볼 수 있다. 예를 들어 CPU 사용량에 대한 정보를 보려면 c를 누르면 되고, 메모리는 m, 디스크는 d, 네트워크는 n 등을 누르면 된다.

CPU 사용량에 대한 정보를 확인하기 위해 c를 누르면 다음과 같은 화면이 출력되면서 일정한 간격으로 CPU사용량에 대한 정보를 계속 확인해 볼 수 있다.

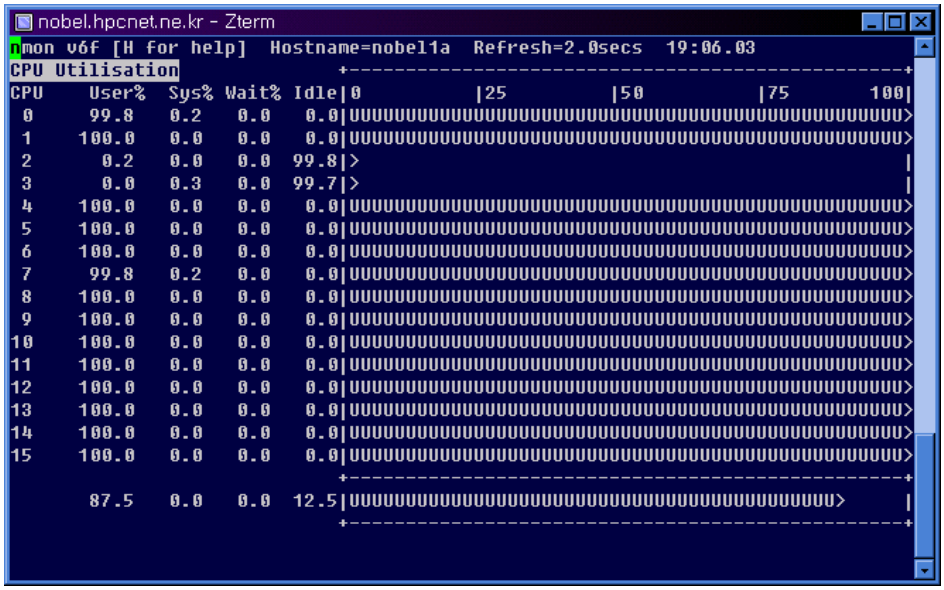


그림 6.2 cpu 사용량 체크

터미널 창의 세로 크기를 좀 더 늘린 후 t를 누르면 top 프로세스에 대한 정보를 같이 볼 수 있다.

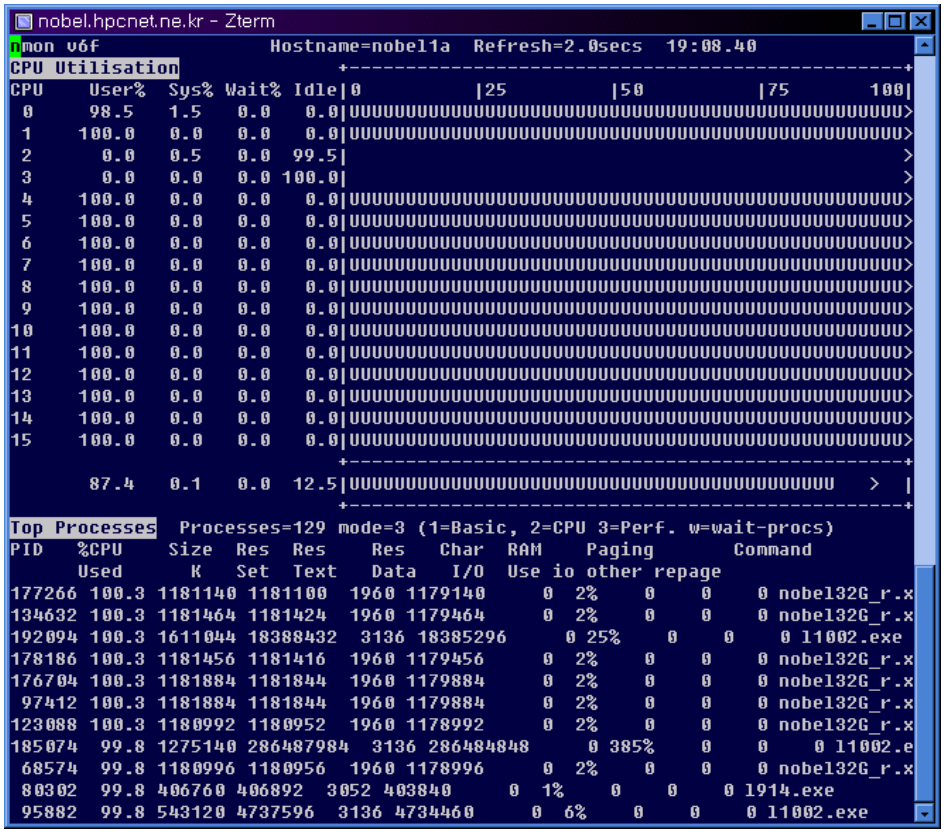


그림 6.3 프로세스 정보

여기서 사용된 각 키는 그에 해당하는 정보를 보여주는 것에 대한 on/off 키와 같은 역할을 하기 때문에 한번 더 같은 키를 눌러 주면 그 정보를 표시하지 않게 된다. 즉 위에서 c를 한번 더 눌러 주면 다음과 같이 표시 된다.

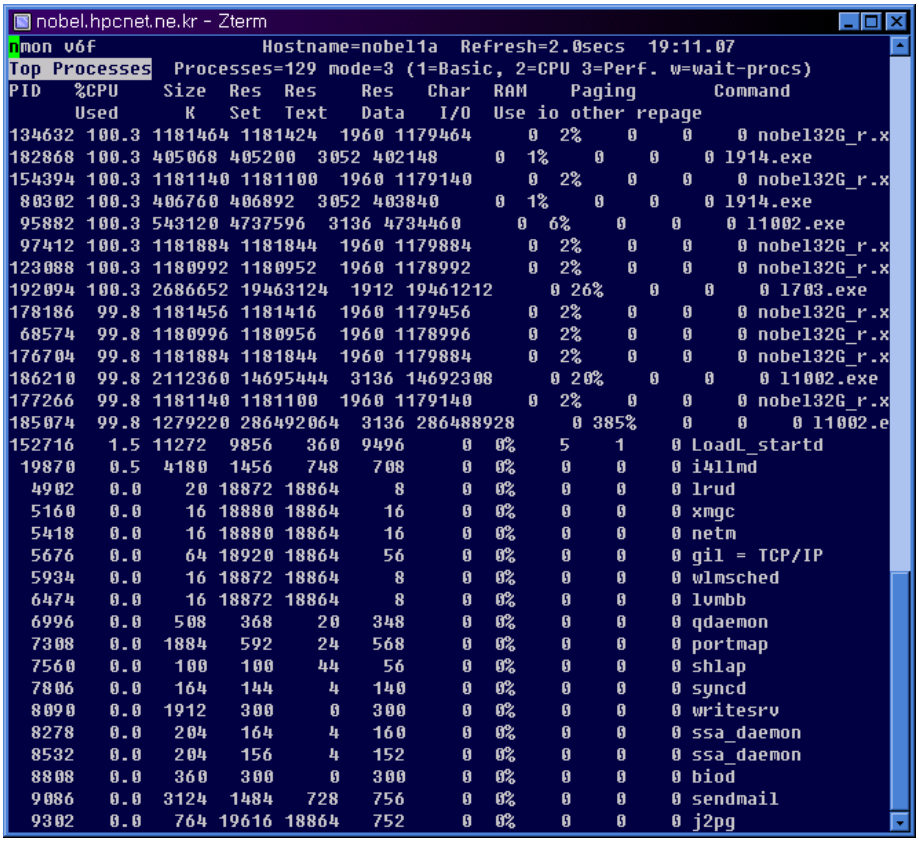


그림 6.4 프로세서 정보2

즉 CPU 사용량에 대한 정보가 없어지고 top 프로세서에 대한 정보만 보여주게 되는 것이다. 이와 같은 방법으로 초기화면에 보여준 원하는 정보에 대한 키를 이용하여 실시간으로 모니터링을 할 수 있다. nmon을 끝낼 때는 q를 눌러주면 된다.

7. CVS

7.1 CVS 소개

CVS(Concurrent Versions System, 버전관리 시스템)는 각종 코드 및 파일의 버전을 쉽게 관리할 수 있도록 도와주는 유틸리티이다. 이러한 버전 관리 시스템은 큰 프로그래밍 프로젝트에서 일반적으로 사용되며 개인 파일 관리를 위해서도 사용된다. 개인 사용자인 경우 그 필요성을 못 느낄 수도 있으나 코드를 작성하고 디버깅하는 과정이 자주 있을 경우 그 편리함을 많이 누릴 수 있다.

예를 들어 간단한 프로그램을 하나 작성한다고 하자. 비교적 간단해 보이는 문제여서 곧바로 코딩을 시작하여 하루만에 작업을 마쳤다. 하지만 실행해보니 의도한 바대로 결과가 나오지 않는다. 원인이 무엇인지 알아내기 위해 이부분 저부분을 뜯어고치다 보면 처음의 모습은 거의 찾아볼 수 없는 복잡한 코드로 변해버린다. 하지만 결국 발견하는 것은 사소한 실수, 이제 그 동안의 작업을 모두 원래대로 돌려놓으려 하지만 이것 역시 버그를 잡는 일 못지 않게 어려운 일이다. 나름대로 바뀌었다고 생각되는 부분을 복구해봐도 제대로 되지 않고, 결국에는 처음부터 다시 코딩해 버린다. 만일 이 때 처음의 코드를 백업해 놓았다면 찾아낸 버그만을 고치고 작업을 계속 진행할 수 있을 것이다. 하지만 그렇다고 해서 매 작업 단계마다 코드를 백업해 놓는다면 조금만 지나도 수없이 많은 파일들이 생겨날 것이다. 이런 파일들은 공간도 많이 차지할 뿐더러 그 많은 파일들을 관리하는 것 자체가 또다른 문제가 될 것이다. CVS는 바로 이러한 문제를 해결해 준다. CVS를 이용하면 매 작업 단계마다 코드를 저장할 수 있음은 물론, 원하는 단계의 코드를 언제라도 꺼내 볼 수 있다. 게다가 CVS는 각 단계에서 변경된 부분만을 저장하기 때문에 저장 공간도 많이 필요로 하지 않는다.

또한 각 단계마다 작업한 내용을 글로 적어 함께 저장할 수 있으므로 개발 내용을 한눈에 알아볼 수 있게 해 준다.

CVS는 이미 대다수의 공개 프로젝트에서 사용되어 그 효능이 입증되었다. 실제로 우리가 알고 있는 대부분의 공개 프로젝트가 CVS를 사용한다. Apache HTTP server, Mozilla 등이 대표적인 예이다. 이러한 공개 프로젝트들은 대부분 현재 개발 중인 내용을 CVS을 통해 모든 사람들이 받아 볼 수 있도록 하고 있다.

7.2 동작 방식

CVS는 중앙에 저장소를 두어 파일을 저장하고, 모든 사용자가 파일에 엑세스할 수 있도록 설정한다. 이는 특별한 데이터베이스나 다른 저장 매체를 사용하는 것이 아니라 일반적인 디렉토리를 하나 두고 이곳을 저장소로 이용하게 된다. 이 저장소에는 CVS의 전반적인 설정 사항과 각 프로젝트의 파일들(문서, 프로그램 등)은 물론, 각 파일의 버젼 관리에 필요한 정보, 파일별 작업 기록들을 저장하게 된다. 여기에 있는 파일들은 모두 CVS가 관리하므로 사용자는 이 파일들을 직접 건드릴 필요는 없다.

저장소에 프로젝트를 위한 공간이 마련되고 나면 사용자들은 저장소의 내용을 가지고 직접 작업할 수는 없으므로 우선 저장소에 있는 내용을 복사하여 자신의 작업 디렉토리를 만들어야 한다. 이 과정을 CVS에서는 **checkout**이라고 한다. checkout으로 만들어진 작업 디렉토리는 자신만의 작업 공간이 되며, 개발자는 그 파일들을 마음대로 변경할 수 있다. 이후로는 계속 작업을 진행하면서 CVS의 명령을 이용하여 자신의 작업 결과를 저장소로 옮기고, 다른 사람이 저장소에 올려 놓은 작업 결과를 받아오는 일만을 반복하면 된다. 공동 작업으로 인해 생길 수 있는 문제들의 대부분은 CVS가 해결해 줄 것이다.

7.3 저장소(Repository) 설정

7.3.1 초기화

CVS를 사용하기 위해서 가장 먼저 할 일은 각 프로젝트의 파일들을 저장할 저장소(Repository)의 위치를 정하는 것이다. 저장소는 다음과 같이 초기화할 수 있다.

```
$ cvs -d /system/center/zootun/cvs init
```

여기서 -d는 저장소의 위치를 나타내며 위와 같은 경우 /system/center/zootun/cvs 밑에 저장소가 만들어지게 된다. 이 명령을 수행하면 /system/center/zootun/cvs라는 디렉토리가 생기고, 또한 /system/center/zootun/cvs/CVSROOT 디렉토리가 생성되면서 이 디렉토리 안에는 여러가지 설정 파일들이 위치하고 있게 된다. 저장소(/system/center/zootun/cvs)의 파일들을 직접 수정하는 것은 절대로 피해야 하며 이 파일들을 변경하고자 할 경우 cvs 명령을 이용해야 한다.

7.4 사용방법

CVS를 이용하는 모든 과정은 cvs 명령을 통해 이루어지는데 cvs 명령의 기본 형식은 다음과 같다.

```
Usage: cvs [cvs-options] command [command-options-and-arguments]
 where cvs-options are -q, -n, etc.
 where command is add, admin, etc.
where command-options-and-arguments depend on the specific command
```

7.4.1 저장소 이용

모든 CVS 명령은 저장소의 위치를 알아야 수행될 수 있다. 그래서 다음과 같이 저장소의 위치를 CVSROOT 환경변수를 이용하여 사용자 profile 파일(.profile)에 넣어 준다.

```
CVSROOT=/system/center/zootun/cvs
export CVS
혹은 export CVSROOT=/system/center/zootun/cvs
```

csh인 경우 다음과 같이 할 수 있다.

```
setenv CVSROOT /system/center/zootun/cvs
```

7.4.2 프로젝트 초기화

현재 자신의 코드가 위치한 곳이 /system/center/zootun/orig_src라는 디렉토리이고 현재 다음과 같이 2개의 파일이 이 디렉토리에 있다고 하자.

```
$ pwd
/system/center/zootun/orig_src
$ ls
host.list      parallel_pi.f
```

저장소에 새 프로젝트를 만들고 이 두 파일을 저장하기 위해서는 import 명령을 사용한다. import 명령의 사용법은 다음과 같다. 주의할 점은 이 명령은 반드시 orig_src 디렉토리에서 실행해야 하며 상위 디렉토리에서 수행시 무한루프에 빠지게 된다.

```
$ cvs -d /system/center/zootun/cvs import -m “메시지” 프로젝트이름 vendor_tag release_tag
```

“메시지”는 프로젝트를 시작하면서 저장소에 기록하고 싶은 내용을 적어주면 된다. CVS는 파일을 저장할 때마다 메시지를 적도록 하고 있다. 이 내용을 원하는 때에 다시 볼 수 있으므로 사용자가 파일의 변경 내용을 파악 하는 데에 도움을 줄 수 있다. 뒤의 두 태그는 지금 단계에서는 별 의미가 없으므로 적당한 말을 써 주면 되는데 이때 주의할 점은 태그에서는 .(dot, 마침표)를 사용할 수 없다는 것을 알아 두어야 한다. para_pi라는 프로젝트를 저장소에 만들기 위해서는 다음 명령을 사용한다.

```
 $ cvs -d /system/center/zootun/cvs import -m "Starting para_pi project"
para_pi parallel_pi ver_1
 N para_pi/host.list
 N para_pi/parallel_pi.f

 No conflicts created by this import
```

여기서 CVSROOT가 /system/center/zootun/cvs로 설정되어 있기 때문에 “-d /system/center/zootun/cvs” 부분은 생략해도 상관없다(다음 예부터는 CVSROOT가 /system/center/zootun/cvs로 설정되어 있다고 가정하도록 하겠다.). 위의 명령으로 인해 CVS 저장소에 para_pi이라는 디렉토리가 생성이 되고, 이 디렉토리에 host.list parallel_pi.f 파일을 저장한다. 또한 옵션으로 준 메시지와 각 파일의 부가적인 정보를 기록하게 된다. 출력되는 메시지에서 각 라인의 앞부분에 있는 N은 새로운 파일이 추가된 것을 의미한다.

import 명령으로 프로젝트를 초기화하고 나면, 즉 파일들을 현재 디렉토리에서 CVS 저장소에 저장하고 난 후에는 더 이상 현재 디렉토리에 있는 파일들을 사용해서는 안되며 현재디렉토리에 있는 파일들을 삭제한 다음 작업하고자 하는 파일들을 CVS 저장소에서 불러들인 후 이 파일들을 이용해야 한다. 또는 다른 작업 디렉토리를 이용하여 작업을 할 수 있다.

7.4.3 프로젝트 진행

7.4.3.1 작업공간 생성

작업을 시작하기 위해서는 파일을 마음대로 변경하고 저장하여 테스트해 볼 수 있는 작업 공간이 필요하다. 저장소에 있는 파일들을 불러와 나만의 작업 공간을 만드는 명령이 checkout이다. mywork이라는 디렉토리를 만들어 앞에서 생성된 para_pi를 checkout(혹은 co) 명령으로 불러 보면 다음과 같다.

```
$ pwd
/system/center/zootun/mywork
$ cvs checkout para_pi
cvs checkout: Updating para_pi
U para_pi/host.list
U para_pi/parallel_pi.f
$ ls
para_pi
$ cd para_pi
$ ls
CVS            host.list      parallel_pi.f
```

위와 같이 CVS 저장소로부터 para_pi라는 디렉토리가 로드되며 이 디렉토리를 살펴보면 프로젝트를 초기화했던 파일들이 그대로 들어 있음을 알 수 있다. 또한 추가적으로 CVS라는 디렉토리가 생성되어 있는데 이 디렉토리에는 파일들을 관리하는데 필요한 정보들 즉 각 파일들의 버전, 최종수정시각, 저장소의 위치 등의 내용들이 기록되어 있다. 사용자들은 새로 생성된 para_pi라는 작업 공간에서 파일들을 수정하거나 업데이트를 한 후 이를 다시 저장소로 저장하면 되는 것이다.

7.4.3.2 작업 내용 저장(commit)

이제 생성된 작업 공간에서 작업을 시작할 수 있다. parallel_pi.f 파일을 다음과 같이 수정한 후 저장해 보도록 하자.

```
      pi_sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         pi_sum = pi_sum + 4.d0/(1.d0+x*x)
 20   continue

-------------------->

      pi_sum  = 0.0d0
c$omp parallel do private(i,x) reduction(+:pi_sum)
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         pi_sum = pi_sum + 4.d0/(1.d0+x*x)
 20   continue
```

즉 parallel_pi.f 파일에서 mpi로 병렬화된 부분에 추가로 openmp directive를 사용한 공유메모리 병렬화를 적용하여 hybrid 코드로 만들었다. 이제 컴파일한 후 실행을 해 보고 예상대로 실행된다면 작업 내용을 저장소에 저장하면 된다. 파일의 변경 사항을 저장소에 저장하는 명령은 commit이다.

```
$ cvs commit -m "openmp directive was added" parallel_pi.f
Checking in parallel_pi.f;
/system/center/zootun/cvs/para_pi/parallel_pi.f,v  <--  parallel_pi.f
new revision: 1.2; previous revision: 1.1
done
```

이 명령으로 parallel_pi.f는 “openmp directive was added”란 메시지와 함께 저장소에 저장된다. 그렇다고 해서 이전의 parallel_pi.f가 없어지는 것은 아니며 CVS는 각 버전의 변경 내용을 파악하여 언제라도 사용자가 원하는 버전을 꺼내 줄 수 있도록 파일들을 저장한다. 출력 결과를 보

면 원래 있던 parallel_pi.f는 1.1이고, 새로 저장된 parallel_pi.f는 1.2임을 알 수 있다. 이 번호는 CVS가 자동으로 붙이는 것이면 변경 사항이 저장될 때마다 올라가므로 각 파일마다 다를 수 있다. 나중에 특정 버전이 필요하면 이 번호를 이용해서 불러 오면 된다. commit할 때 마지막에 파일 이름을 주지 않으면 CVS가 변경된 파일을 모두 찾아 저장소에 저장한다. 이때 저장되는 모든 파일에는 같은 메시지가 붙게 되기 때문에 각 변경된 파일마다 다른 메시지를 붙이고 싶다거나 특정 파일의 변경 내용만을 저장하고 싶다면 위의 예처럼 파일 이름을 명시해 주어야 한다.

commit을 수행할 때는 제대로 동작하는지를 확인한 다음에 수행해야 하며 특히 이는 여러 사람이 같이 작업하는 공동 작업에서 더욱 더 중요하다.

7.4.3.3 저장소의 파일 받아오기(update)

자신이 작업한 내용을 모두 저장한 상태라면 작업 디렉토리를 지워버리고 checkout 명령으로 새로운 작업 공간을 만들어 사용할 수 있다. 그러나 만약 파일들이 아주 많고 특히 아주 큰 파일들을 포함할 경우 checkout 명령으로 불러오는데 상당한 시간이 걸릴 수 있다.

이럴 경우 update 명령을 사용하면 된다. 작업중이던 디렉토리에서 이 명령을 수행하면 CVS에 저장된 파일들 중 내가 받아온 이후로 변경된 것들만을 다시 받아 온다.

```
$ cvs update
cvs update: Updating .
```

현재 작업 공간에 있는 파일들이 저장소에 있는 파일과 모두 동일하면 위와 같이 간단한 메시지와 함께 끝나게 된다. 만일 현재 디렉토리에서 parallel_pi.f 파일을 수정하는 중에 update 명령을 수행하면 다음과 같은 메시지를 보여주게 된다.

```
$ cvs update
cvs update: Updating .
M parallel_pi.f
```

parallel_pi.f가 저장소에 있는 것과 다르며 현재 작업공간의 파일이 변경된(Modified) 상태임을 보여준다. 즉 파일의 내용이 원래 저장소에 있는 것과 달라졌을 때 M을 써서 표시하게 된다.

또한 현재 작업 공간에서 파일을 편집하여 수정하는 중에 다른 사용자가 어떤 작업이나 내용을 변경하여 저장소에 저장했을 때, 혹은 사용자 자신이 다른 작업 공간이나 다른 시스템에서 어떤 작업이나 내용을 변경하여 저장소에 저장했을 때 현재 작업 공간에서 수정 후 update를 하게 되면 다음과 같은 메시지가 보여지게 된다.

예를 들어 다른 사용자 혹은 사용자 자신이 다른 작업 공간이나 다른 시스템의 A라는 작업 공간에서 parallel_pi.f를 불러들여 이를 수정한 후 commit을 수행했을 할 경우 다음과 같이 진행 될 것이다.

```
      time2=rtc()
      write(6,*) 'Elapsed time = ',time2-time1

 ----------------→              Standard output 에 my rank 를 출력하도록
추가하고 주석을 달아줌.

      time2=rtc()
 c    my rank added
      write(6,*) 'my rank = ',myid,'Elapsed time = ',time2-time1
```

```
$ cvs commit -m "my rank was added in the STDOUT" parallel_pi.f
Checking in parallel_pi.f;
/system/center/zootun/cvs/para_pi/parallel_pi.f,v  <--  parallel_pi.f
new revision: 1.7; previous revision: 1.6
done
```

이런 상황에서 사용자 자신의 원래 작업 공간인 B 작업 공간에서 동일한 parallel_pi.f 파일을 불러들려 이를 수정한 후 update를 수행할 경우 다음과 같은 메시지를 보여주게 될 것이다.

```
      time2=rtc()
      write(6,*) 'Elapsed time = ',time2-time1

 ----------------→          Standard output 에 my rank 를 출력하도록 한줄
추가하고 주석을 달아줌.

      time2=rtc()
 c    my rank added
      write(6,*) 'my rank = ',myid
      write(6,*) 'Elapsed time = ',time2-time1
```

```
$ cvs update
cvs update: Updating .
RCS file: /system/center/zootun/cvs/para_pi/parallel_pi.f,v
retrieving revision 1.6
retrieving revision 1.7
Merging differences between 1.6 and 1.7 into parallel_pi.f
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in parallel_pi.f
C parallel_pi.f
```

즉 동일한 parallel_pi.f 파일을 A와 B의 작업 공간에서 각각 불러들여 서로 다르게 수정한 상태에서 A 작업공간에서 먼저 commit을 수행한 후 B 작업 공간에서 update를 수행하는 상황이다. 현재 디렉토리의 parallel_pi.f는 1.6에서 변경된 상태이고 저장소에 저장되어 있는 것은 1.4이므로 CVS는 저장소에 있는 1.3 버전과 1.4 버전의 차이를 살펴 보게 된다. 그런 후 서로 충돌하는 내용이 있음을 알고 충돌(C)이 생겼다는 것을 표시하면서 수행을 마치게 된다. 이제 다시 B작업 공간에서 parallel_pi.f파일을 열어 보면 어디서 충돌이 생겼는지를 알수 있게 된다.

```
      time2=rtc()
<<<<<<< parallel_pi.f
c     my rank added
      write(6,*) 'my rank = ',myid
      write(6,*) 'Elapsed time = ',time2-time1
=======
c     my rank added
      write(6,*) 'my rank = ',myid,'Elapsed time = ',time2-time1
>>>>>>> 1.7
```

<<<<<<<와 >>>>>>> 사이가 충돌이 일어난 부분이다. 그 부분은 다시 두 부분으로 나뉘는데, ======= 이전까지가 현재 작업공간에 있는 파일의 내용이고, 그 이후가 저장소에 있는 파일의 내용이다. 사용자는 이걸 보고 어느 한 쪽을 없애거나 두 내용을 적절히 합친 후 다시 commit을 해 주면 된다. 이러한 점은 CVS의 강력한 장점이라고 할 수 있다.

7.4.3.4 파일의 추가/삭제(add/delete, remove)

작업 중 새로운 파일을 추가하거나 삭제할 경우 add와 delete 혹은 remove 명령을 사용한다. 현재 작업공간에서 run.sh 이라는 작업 수행 스크립트를 만들어 CVS에 추가하고자 할 경우 먼저 run.sh이라는 파일을 생성한 후 다음과 같이 하면 된다.

```
$ cvs add run.sh
cvs add: scheduling file `run.sh' for addition
cvs add: use 'cvs commit' to add this file permanently
```

출력부분에 있듯이 add 명령만으로는 저장소에 run.sh이라는 파일이 생기지 않고 commit 명령을 수행해 주어야 한다. add 명령은 단지 commit 명령시에 run.sh을 추가해야 한다는 것을 기록해 놓은 것뿐이다. 이는 update 명령으로도 확인이 가능하다.

```
$ cvs update
cvs update: Updating .
A run.sh
```

추가될 파일임을 의미하는 A(added)를 표시하게 된다. commit 명령을 이용하여 run.sh 파일을 저장소에 저장해 보면 다음과 같다.

```
$ cvs commit -m "new file(run.sh) added" run.sh
RCS file: /system/center/zootun/cvs/para_pi/run.sh,v
done
Checking in run.sh;
/system/center/zootun/cvs/para_pi/run.sh,v  <--  run.sh
initial revision: 1.1
done
```

이를 다른 작업 공간에서 update를 하게 되면 다음과 같이 수행되면서 run.sh파일이 생기게 된다.

```
$ cvs update
cvs update: Updating .
U run.sh
$ ls
CVS           host.list      parallel_pi.f  run.sh
```

delete(혹은 remove) 명령을 통해 파일을 삭제하고자 할경우에도 add와 비슷하게 수행하면 된다. run.sh 파일을 삭제 하고자 할 경우 먼저 현재 작업 공간에서 run.sh파일을 삭제한 후 delete(혹은 remove) 명령을 수행하고 commit 명령을 실행하면 된다.

```
$ rm run.sh
$ ls
CVS            host.list      parallel_pi.f
$ cvs delete run.sh
cvs remove: scheduling `run.sh' for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs commit -m "run.sh file deleted" run.sh
Removing run.sh;
/system/center/zootun/cvs/para_pi/run.sh,v  <--  run.sh
new revision: delete; previous revision: 1.1
done
```

7.4.3.5 작업 기록 열람(log)

CVS의 작업 내용을 즉 commit 명령으로 저장할때마다 입력한 메시지 내용을 보고자 할경우 log 명령을 사용하면 된다. parallel_pi.f 파일에 대한 내용을 볼려면 다음과 같이 수행하면 된다.

```
$ cvs log parallel_pi.f

RCS file: /system/center/zootun/cvs/para_pi/parallel_pi.f,v
Working file: parallel_pi.f
head: 1.7
branch:
locks: strict
access list:
symbolic names:
        ver_1: 1.1.1.1
        parallel_pi: 1.1.1
keyword substitution: kv
total revisions: 8;     selected revisions: 8
description:
----------------------------
revision 1.7
date: 2004/05/07 01:38:31;  author: zootun;  state: Exp;  lines: +2 -1
my rank was added in the STDOUT
```

```
----------------------------
revision 1.2
date: 2004/05/04 02:08:27;  author: zootun;  state: Exp;  lines: +1 -0
openmp directive was added
----------------------------
revision 1.1
date: 2004/05/04 01:46:34;  author: zootun;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
revision 1.1.1.1
date: 2004/05/04 01:46:34;  author: zootun;  state: Exp;  lines: +0 -0
Starting para_pi project
=============================================================
```

commit을 수행할 때마다 그에 해당하는 버전과 날짜와 시각, 저장한 id, 메시지 등이 기록되어 있는 것을 알 수 있다. 이렇게 log 명령을 쓰면 언제 누가 어떤 작업을 했는지 알 수 있다. 이 기록내용 중의 한 버전을 현재 작업 공간에 가져와서 보고 싶다면 다음과 같이 수행해 주면 된다.

```
$ cvs update -r 1.2 parallel_pi.f
U parallel_pi.f
```

이는 log 명령의 결과에 있는 내용 중 revision 1.2에 해당하는 parallel_pi.f 파일을 현재 작업 공간에 가져오게 된다. 이는 현재 파일이 잘못 수정되어 버그가 발견되었거나 다시 과거의 파일로 돌아가고자 할 경우 유용하게 이용될 수 있으며 이 또한 CVS의 강력한 장점중 하나라고 하겠다.

참고 http://home.bawi.org/~minskim/moin.cgi/CVS%20%EC%82%AC%EC%9A%A9<

ISBN 89-5884-314-4 93560

Optimization Tools for High Performance Computing