Contents

ADAPTOR
Users Guide
Version 1.0 (June 1993)

T. Brandes

Internal Report No. Adaptor 2

June 30, 1993

High Performance Computing Center
German National Research Institute for Computer Science
P. O. Box 1316
D-5205 Sankt Augustin 1
Federal Republic of Germany
Tel.: +49 (0)2241 / 14-2492
E-mail: Thomas.Brandes@gmd.de

ADAPTOR
Users Guide
Version 1.0 (June 1993)

T. Brandes

German National Research Center for Computer Science,
P.O. Box 1316, D-5205 Sankt Augustin 1, FRG

Abstract

ADAPTOR (Automatic DAta Parallelism TranslatOR) is a tool for transforming data parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing. The input language is very similar to CM Fortran and High Performance Fortran though not all features of these language are supported.

The generated message passing programs will run on different multiprocessor systems with distributed memory, but also on shared or virtual shared memory architectures.

In this paper it is described in which way this transformation tool can be used either interactively or in a batch version, and how to run the generated parallel message passing programs.

1 Overview

The Adaptor tool system consists of a source-to-source transformation (fadapt) and a basic set of routines for message passing and operations on sequential and distributed arrays (DALIB).

The source-to-source transformation can be done in a batch manner or interactively. In the interactive version a graphical environment allows the user to select units of the source program (program, functions, subroutines) or variables in a unit to get information about them.

After the transformation process is done the generated programs have to be compiled and linked. The distributed array library (DALIB) will be linked together with the compiled sources to executable programs. These executables can easily be started on the parallel system and will hopefully run faster.

Before continuing make sure that Adaptor has already been installed on your machine [Bra93a]. Though you can make your own installation, it is recommended to make a system-wide installation for all users.

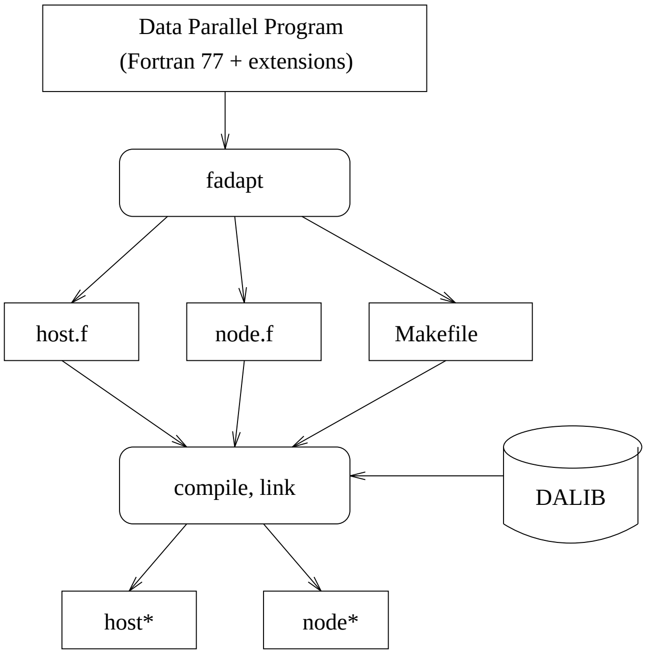


Figure 1: Overview of Adaptor

2 Setting the Environment

Let be `<install-dir>` the directory on the machine where Adaptor is installed. Every user should set a link to this directory in the following way:

```
cd                          ! change to home directory
ln -s <install-dir> adaptor
```

Furthermore, the `bin`-directory of Adaptor should be included in the path variable, the `man`-directory in the manpath variable.

```
setenv PATH $HOME/adaptor/bin:$PATH
setenv MANPATH $HOME/adaptor/man:$MANPATH
```

Now there should be no problems to call the commands of Adaptor (fadapt, fstrip, adapt.clean). A manual page for 'fadapt' is available.

The following commands should now work.

```
fadapt -help
man fadapt
```

3 Batch Use of Adaptor

In most cases it is sufficient to call the batch version of Adaptor. So it is possible to make a direct source-to-source translation without the usage of X-Windows.

3.1 Call of Adaptor

With the following command the sources of a data parallel program are directly translated to a message passing program.

```
fadapt <source_file_names>
```

In the current version more than one source file may be specified, but only for the batch use. All soure files together stand for one data parallel program. There must be exactly one main program in the specified source files. All subroutines and functions that are not defined are assumed to be external.

If no source file is specified, the interactive version will be called.

If the translation is executed correctly, the following files will have been generated:

```
    host.f, node.f    (host and node program)
or  cube.f            (only node program)
or  node1.f           (single node program)
and Makefile          (Makefile for compiling and linking)
```

Furthermore, some files will be created which contain analysis informations about the translation process. These files are very useful in situations where errors have been encountered. If an error within one phase occurs, the translation will stop directly after this phase.

```
adaptor.def      (protocol file of phase 1 of semantic analysis)
adaptor.sem      (protocol file of phase 2 of semantic analysis)
adaptor.cf       (protocol file of phase 3 of semantic analysis)
adaptor.anal     (protocol file of adaptor phase 1: analysis)
adaptor.dist     (protocol file of adaptor phase 2: distributions)
adaptor.temps    (protocol file of adaptor phase 3: temporary variables)
adaptor.init     (protocol file of adaptor phase 4: initial translation)
adaptor.seq      (protocol file of adaptor phase 5: serialization)
adaptor.trans    (protocol file of adaptor phase 6: final translation)
unparse.f        (source program before final translation)
```

These files can be deleted with the command adapt.clean1.

3.2 Choosing the Target Machine

The following target machines are supported:

- Intel iPSC/860
- Sun workstation net (using PVM)
- IBM Risc workstation net (using PVM)
- KSR 1
- CM 5
- Parsytec GC, GCel (using PARIX)
- Meiko CS1, Meiko CS2
- Silicon Graphics multiprocessor machines
- Alliant FX/2800

The following options can be used to specify the desired target machine. Only the last option will be the valid one.

```
[-sun]          target machine = SUN4, PVM
[-ibm]          target machine = IBM, PVM
[-alliant]      target machine = Alliant FX/2800
[-all_pvm]      target machine = Alliant FX/2800 with PVM
[-ipsc]         target machine = iPSC/860
[-gc]           target machine = Parsytec GC
[-meiko_cs1]    target machine = Meiko CS 1
[-meiko_cs2]    target machine = Meiko CS 2
[-ksr]          target machine = KSR 1
[-ksr_pvm]      target machine = KSR 1 with PVM
[-sgi]          target machine = Silicon Graphics (IRIX)
[-cm]           target machine = CM 5
```

The new generated source programs are independent of the target machine. Only some syntax might be machine-dependent.

In contrast, the `Makefile`, that is also generated, depends on the selected target machine.

3.3 Choosing the Execution Model

The user can select between the following three programming models:

- `HOST-NODE` will generate a host program (`host.f`) and a node program (`node.f`). The node program runs on all available nodes of the parallel machine, while the host program contains all I/O operations that will be executed on the front end system.
- `ONLY-NODE` will generate a program (`cube.f`) that runs on all available nodes. There is no host program. The first node takes care of all I/O operations.
- `UNI-PROC` will generate a program (`node1.f`) that runs on a single node. It has no communication and therefore it ought to be faster than the previous one running on a single node.

For the call of Adaptor the model can be specified by the following options:

```
-H         model = HOST_NODE   (host and node program)
-N         model = ONLY_NODE   (only node program)
-1         model = UNI_PROC    (single node program)
-uniproc   model = UNI_PROC, same as -1
```

3.4 Handling of Distributed Static Arrays

Big static arrays are usually distributed among the available nodes. The size of a part of the array on one node should be the size of the original array divided by the number of processors. Otherwise one might run in memory problems.

But if the compiler of the target machine does not support dynamic arrays, the size of the distributed array on one node must be fixed. Therefore the translation has to know about the minimal number of processors that will be available when running the generated message passing program.

This number of minimal processors can be specified with the -p option.

```
[-p <n>]           Minimal number of processes is n
```

The effect resulting of this option is the following one:

```
real A(1000)    ! A is block distributed
```

becomes in the new generated program:

```
real A(1000)        ! for -p 1
real A(334)         ! for -p 3
real A(100)         ! for -p 10
real A(:)           ! if target compiler supports dynamic arrays
```

The parallel program must not run on a parallel machine with less processors that has been specified with the number of minimal processors, otherwise there will be severe run-time errors. But it is possible to use more processors. The translation with $-p$ 1 ends up in a parallel program that runs for any number of processors, but it might cause some memory problems.

3.5 Handling of Dynamic Arrays

Adaptor supports dynamic arrays like allocatable and automatic arrays. Sometimes this feature is not available for the compiler of the target machine. In this case the dynamic arrays are translated to static arrays with a given size.

```
-D        (generated programs will have dynamic arrays)
-S size   (dynamic arrays are translated to
           static arrays with default size)
```

The effect of this option is the following one:

```
real A(:)
```

becomes in the new generated program:

```
real, allocatable :: A(:)   !  for -D
real A(:)                   !  for -D and some target machines
real A(25000)               !  for -S 25000
```

3.6 Translation of Array Operations

Adaptor supports array syntax. Sometimes this feature is not available for the compiler of the target machines. In this case the array operations must be translated to sequential loops of Fortran 77.

```
-F90        Target Language knows about array operations
-F77        Arrays operations are translated to loops
```

3.7 Default Distributions

If the user does not give any layout or distribution directive for an array in his data parallel program, a default distribution will be chosen. The following options are intended for selecting a strategy for the default distribution.

```
-ddr      default distribution of arrays is replicated
-ddb      default distribution of arrays is block distribution
          along the last dimension
-ddcm     default distribution depends on the use of the array,
          the same rules as in CM Fortran will apply
```

Scalar variables will always be replicated.

3.8 Setting the Installation Directory

The installation directory must contain the help file and the DALIB. It is possible to refer to the correct directory if there are some inconveniences (e.g. if the Adaptor system is mounted from another workstation).

```
-home <dir>     Home Directory
```

3.9 Splitting the Sources

Sometimes it is quite useful to split the generated message passing programs up into source files for every unit. One reason may be that compilers cannot compile large source codes, whereas another reason might be that one wants to take advantage of a parallel 'make'.

The splitting itself is done by using the 'fsplit' command that is usually available for every Fortran compiler.

With the following options it can be specified whether splitting should be done or not.

```
-split        ! generated programs will be split up
-nosplit      ! only one source file
```

3.10 Changing Default Options

With the following command one can get a summary of the default options.

```
fadapt -defaults
```

An alias can be defined to set its own default options.

```
alias fadapt1 fadapt -S 1000000 -F77 -ksr -N !*
```

Afterwards the command fadapt1 can be used like fadapt but with different default options.

```
fadapt1 -defaults

Defaults of fadapt:
===================

Home Directory       : /home/brandes/adaptor
Default Distribution : arrays are block distributed (ddb)
Target Machine       : Kendall Square Research 1
Target Model         : N=Only Nodes
Target Language      : F77 (Fortran 77)

Dynamic Arrays will be          : S=static with minimal size = 1000000
Minimal number of processes (p) : 1

Generated sources will not be split (nosplit)
```

3.11 Other Options

For the batch translation the user has the following possibilities:

```
fadapt [options] ( -parse | -semantic | -call | -adapt ) <filenames>

    parse    : only syntactical analysis
    semantic : syntactical and semantical analysis
    call     : semantic + generating a call graph (test.call)
    adapt    : full source to source translation
```

A detailed description of the input language for Adaptor can be found in [Bra93b]. There exist also many example programs that should be used to analyze the functionality of the transformation tool.

4 Examples for the Translation

This section shows for one example program how the translation works and what can be done with some different options.

4.1 The Input Program

The following data parallel program computes the number of primes in the range from 2 to n. The program uses dynamic arrays and array syntax. Timing functions are used to measure the run time of the program.

There is exactly one array in the program. This array will be distributed among the nodes.

```
      program prime
      integer n, s, k
      logical*1 a(:)
!hpf$ distribute a(block)
      print *, 'Input n for counting primes in range 2 to n : '
      read *, n
      allocate (a(1:n))
      call cm_timer_clear (1)
      call cm_timer_start (1)
      a = .true.
      a(1) = .false.
      k = 2
      do while (k*k .le. n)
         a(k*k:n:k) = .false.
         k = k + 1
         do while (.not. a(k))
           k = k + 1
         end do
      end do
      s = count (a)
      call cm_timer_stop (1)
      print *, 'There are ',s,' primes until ', n
      call cm_timer_print (1)
      deallocate (a)
      end
```

4.2 Compiling for a Single Node

The following command translates the data parallel program to a sequential Fortran 77 program with static arrays.

```
fadapt -1 -S 1000000 -F77 prime.f
```

In this case the dynamic array becomes a static array of size 1000000. The array operations will be translated to sequential loops.

```
      SUBROUTINE NODEMODULE ()
      INTEGER*4 N
      INTEGER*4 S
      INTEGER*4 K
      LOGICAL*1 A (1:1000000)
      INTEGER*4 A_DIM1
      INTEGER*4 A_OFS
      INTEGER*4 I_1
      PRINT *,'Input n for counting primes in range 2 to n : '
      READ *, N
      A_DIM1 = N
```

```
      A_OFS = 1-1
      IF (A_DIM1 .gt. 1000000) THEN
         PRINT *,'NODEMODULE: A is out of memory, needs : ',A_DIM1
      END IF
      call dalib_clear_timer (1)
      call dalib_start_timer (1)
cdir$ ivdep
      DO I_1=1,N
         A(A_OFS+I_1) = .TRUE.
      END DO
      A(A_OFS+1) = .FALSE.
      K = 2
      DO WHILE (K*K .le. N)
cdir$ ivdep
        DO I_1=K*K,N,K
           A(A_OFS+I_1) = .FALSE.
        END DO
        K = K+1
        DO WHILE ( .not. A(A_OFS+K))
          K = K+1
        END DO
      END DO
      S = 0
cdir$ ivdep
      DO I_1=1,N
         IF (A(A_OFS+I_1)) THEN
            S = S+1
         END IF
      END DO
      call dalib_stop_timer (1)
      PRINT *,'There are ',S,' primes until ',N
      call dalib_print_timer (1)
      END
```

The generated program contains subroutine calls to the DALIB for timing. The program runs only on a single node, so no message passing is required.

4.3 Generating a Host-Node Program

The next command translates the data parallel program to a parallel Fortran 77 host and node program with message passing.

```
fadapt -H -S 1000000 -F77 prime.f
```

The host program contains all I/O-operations. Input values will be broadcast to all nodes. Furthermore, the host program has the same control flow as all nodes. But it has no operations on distributed arrays.

```
      SUBROUTINE HOSTMODULE ()
      INTEGER*4 N
      INTEGER*4 S
      INTEGER*4 K
      LOGICAL*1 A_SC1
      INTEGER*4 I_1
      PRINT *,'Input n for counting primes in range 2 to n : '
      READ *, N
      call dalib_broadcast (N,4,0)
```

```
      call dalib_clear_timer (1)
      call dalib_start_timer (1)
      K = 2
      DO WHILE (K*K .le. N)
        K = K+1
        call dalib_node_get (A_SC1,A_SC1,1,N,K)
        DO WHILE ( .not. A_SC1)
          K = K+1
          call dalib_node_get (A_SC1,A_SC1,1,N,K)
        END DO
      END DO
      S = 0
      call dalib_reduction (S,7)
      call dalib_stop_timer (1)
      PRINT *,'There are ',S,' primes until ',N
      call dalib_print_timer (1)
      END
```

This is the node message-passing program for all nodes of the parallel machine:

```
      SUBROUTINE NODEMODULE ()
      INTEGER*4 N
      INTEGER*4 S
      INTEGER*4 K
      INTEGER*4 A_LOW, A_HIGH
      INTEGER*4 A_START, A_STOP, A_INC
      LOGICAL*1 A (1:1000000)
      INTEGER*4 A_DIM1, A_OFS
      LOGICAL*1 A_SC1
      INTEGER*4 I_1
      LOGICAL*4 dalib_have_i
      EXTERNAL dalib_have_i
      call dalib_broadcast (N,4,0)
      call dalib_array_pardim (N,A_LOW,A_HIGH)
      A_DIM1 = A_HIGH-A_LOW+1
      A_OFS = 1-A_LOW
      IF (A_DIM1 .gt. 1000000) THEN
         PRINT *,'NODEMODULE: A is out of memory, needs : ',A_DIM1
      END IF
      call dalib_clear_timer (1)
      call dalib_start_timer (1)
cdir$ ivdep
      DO I_1=A_LOW,A_HIGH
         A(A_OFS+I_1) = .TRUE.
      END DO
      IF (dalib_have_i(N,1)) THEN
         A(A_OFS+1) = .FALSE.
      END IF
      K = 2
      DO WHILE (K*K .le. N)
        call dalib_local_range (N,K*K,N,K,A_START,A_STOP,A_INC)
cdir$ ivdep
        DO I_1=A_START,A_STOP,A_INC
           A(A_OFS+I_1) = .FALSE.
        END DO
        K = K+1
        call dalib_node_get (A_SC1,A(A_OFS+K),1,N,K)
        DO WHILE ( .not. A_SC1)
          K = K+1
          call dalib_node_get (A_SC1,A(A_OFS+K),1,N,K)
        END DO
      END DO
      S = 0
cdir$ ivdep
```

```
DO I_1=A_LOW,A_HIGH
   IF (A(A_OFS+I_1)) THEN
      S = S+1
   END IF
END DO
call dalib_reduction (S,7)
call dalib_stop_timer (1)
call dalib_print_timer (1)
END
```

4.4 Generating a Only-Node Program

In many cases it is not necessary or useful to have an own host program. The following translation will generate only a node program.

```
fadapt -N -S 1000000 -F77 prime.f
```

There is no separate host program, but the first node will be responsible for the I/O activities.

```
SUBROUTINE NODEMODULE ()
...
LOGICAL*4 dalib_have_i
EXTERNAL dalib_have_i
INTEGER*4 dalib_pid
EXTERNAL dalib_pid
IF (dalib_pid() .eq. 1) THEN
   PRINT *,'Input n for counting primes in range 2 to n : '
   READ *, N
END IF
call dalib_broadcast (N,4,1)
call dalib_array_pardim (N,A_LOW,A_HIGH)
A_DIM1 = A_HIGH-A_LOW+1
A_OFS = 1-A_LOW
IF (A_DIM1 .gt. 1000000) THEN
   PRINT *,'NODEMODULE: A is out of memory, needs : ',A_DIM1
END IF
call dalib_clear_timer (1)
call dalib_start_timer (1)
...
call dalib_reduction (S,7)
call dalib_stop_timer (1)
IF (dalib_pid() .eq. 1) THEN
   PRINT *,'There are ',S,' primes until ',N
END IF
call dalib_print_timer (1)
END
```

4.5 Using Dynamic Arrays

In the previous translations the dynamic array was translated to a static array. If the compiler of the parallel machine supports dynamic arrays, the following translation will be the better solution.

```
fadapt -N -D -F77 prime.f
```

This was the solution with static arrays:

```
      LOGICAL*1 A (1:1000000)
      INTEGER*4 A_DIM1, A_OFS
      ...
      call dalib_array_pardim (N,A_LOW,A_HIGH)
      A_DIM1 = A_HIGH-A_LOW+1
      A_OFS = 1-A_LOW
      IF (A_DIM1 .gt. 1000000) THEN
         PRINT *,'NODEMODULE: A is out of memory, needs : ',A_DIM1
      END IF
```

Here is the generated node program with a dynamic array.

```
      LOGICAL*1 A (:)
      ...
      call dalib_array_pardim (N,A_LOW,A_HIGH)
      ALLOCATE (A(A_LOW:A_HIGH))
```

This program works also for input values greater than 1000000.

4.6 Using Array Operations

The array operations of the data parallel program have been translated to sequential Fortran 77 loops.

```
cdir$ ivdep
      DO I_1=A_LOW,A_HIGH
         A(A_OFS+I_1) = .TRUE.
      END DO
      ...
      K = 2
      DO WHILE (K*K .le. N)
        call dalib_local_range (N,K*K,N,K,A_START,A_STOP,A_INC)
cdir$ ivdep
        DO I_1=A_START,A_STOP,A_INC
           A(A_OFS+I_1) = .FALSE.
        END DO
        K = K+1
        ...
        END DO
      END DO
      S = 0
cdir$ ivdep
      DO I_1=A_LOW,A_HIGH
         IF (A(A_OFS+I_1)) THEN
            S = S+1
         END IF
      END DO
      call dalib_reduction (S,7)
      call dalib_stop_timer (1)
      ...
```

If the compiler of the parallel machine supports array syntax, the following translation will also be possible:

```
fadapt -N -D -F90 prime.f
```

These are the array operations restricted to a single node.

```
cdir$ ivdep
      A(A_LOW:A_HIGH) = .TRUE.
      ...
      K = 2
      DO WHILE (K*K .le. N)
        call dalib_local_range (N,K*K,N,K,A_START,A_STOP,A_INC)
        A(A_START:A_STOP:A_INC) = .FALSE.
        K = K+1
        ...
      END DO
      S = COUNT(A)
      call dalib_reduction (S,7)
      call dalib_stop_timer (1)
      ...
```

5 Interactive Use of Adaptor

The interactive translation tool is realized with Athena widgets based on the X-Window system [NO90, O'R90].

The interactive tool will be called if no source file is specified.

```
fadapt [options]
```

After calling fadapt a window should be displayed on your X-Server. If any problems occur, check whether

- the X-Server is running,
- the environment variable DISPLAY has been set to the address of the machine running the X-Server,

  ```
  setenv DISPLAY hostname:0.0
  ```

- the client (in this case the machine running fadapt) has been authorized to write on the screen of the X-Server (use command `xhost`)

  ```
  xhost + <hostname1>
  ```

5.1 Description of the Window

The window of Adaptor consists of the following areas and lines (see figure 2):

- **GMD Logo**

  All users should never forget where this nice tool has been developed.

- **Command Line**

  In the top line of the window all commands are listed that can be invoked at the actual time. This list varies with each step of the translation.

- **Filename Line**

  In the second line the name of the selected file is displayed.

- **Message Line**

  In the third line the last message is shown that gives information about the success of the last command. Also some help information will be shown here.

- **Edit Area**

  The largest window is an editor where the selected file will be displayed. Scrolling is possible for larger files. The editor is used to highlight actual positions of units and variables. In the current version the edited file cannot be changed (read-only).

- **Unit Area**

  In the Unit Area every unit of the program has an item that can be chosen.

- **Variable Area**

  In the Variable Area every variable of the selected unit has an item that can be selected.

5.2 Help

When a command or label widget is selected with the right mouse button, a help window appears that gives some information for the corresponding command.

The topic for which help is required can also be chosen in a submenu of the $\boxed{\text{Help}}$ command.

The window is released after selecting the $\boxed{\text{Quit}}$ command in the help window.

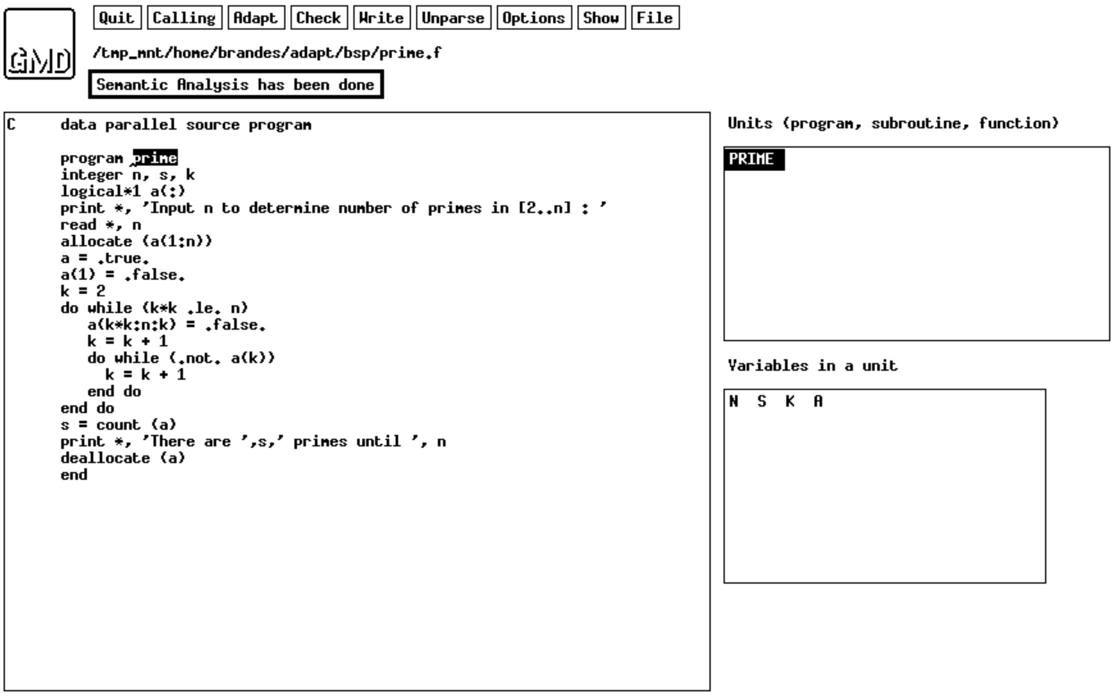


Figure 2: Interactive Environment of Adaptor

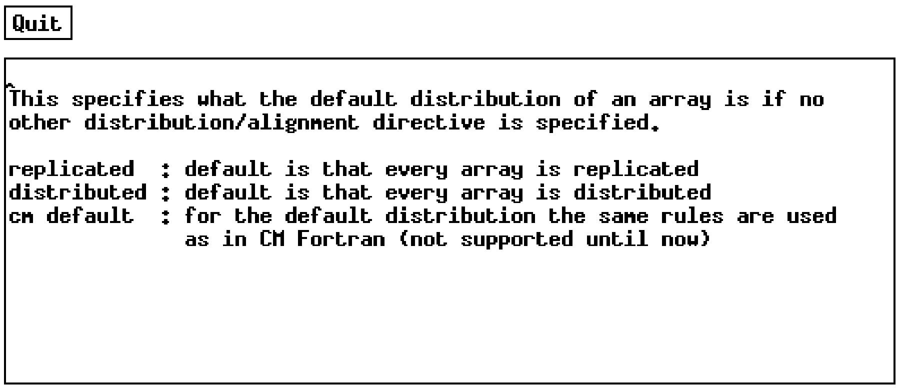


Figure 3: Example of a Help Window

5.3 File Selection

The source file for the translation can be set with the command $\boxed{\text{File}}$. When selecting this button with the mouse all entries of the current directory are popped up in a menu. If the chosen item is a file, this file becomes the current selected file and is displayed in the filename line. If the chosen item is a directory, this directory becomes the new current directory where the file selection can be done again.

It should be observed that only one source file can be translated. Therefore it is necessary that sources of the given program are put together in one file.

5.4 Show

With the command $\boxed{\text{Show}}$ the selected file is shown in the editor window. Only the file in the edit window can be parsed.

This command can be called at every time, but then the file has to be parsed again.

5.5 Parse

When selecting the command $\boxed{\text{Parse}}$ the source file is parsed, and an abstract syntax tree is generated. In the message window it is shown how many errors have been occurred.

If there are any errors, semantic analysis and adaptation cannot be done. In this case Adaptor should be quitted and the source file be edited by using another editor. Afterwards, Adaptor can be restarted.

5.6 Semantic

The semantical analysis will be done after selecting the command $\boxed{\text{Semantic}}$. In this phase the declaration tables are generated. Only the use and declaration of identifiers are checked, but not the correct typing.

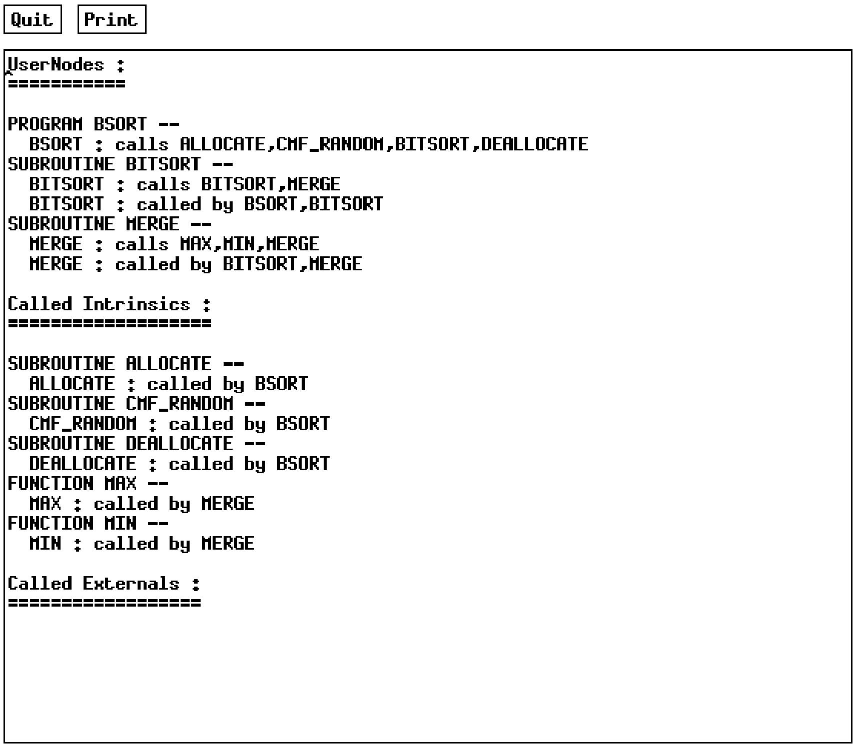
If this phase is successful all units of the source program will get an item in the unit area.

5.7 Calling

When selecting the command $\boxed{\text{Calling}}$ a call graph is generated that is written to the file `test.call`. This file will be displayed with the next command $\boxed{\text{CallGraph}}$ that appears in the command line after the file has been generated.

5.8 CallGraph

With the command CallGraph the file `test.call` is displayed in a new generated window. This file can be printed by selecting the Print command in this window. After viewing the file the command Quit should be selected.



```
Quit  Print

UserNodes :
===========

PROGRAM BSORT --
  BSORT : calls ALLOCATE,CMF_RANDOM,BITSORT,DEALLOCATE
SUBROUTINE BITSORT --
  BITSORT : calls BITSORT,MERGE
  BITSORT : called by BSORT,BITSORT
SUBROUTINE MERGE --
  MERGE : calls MAX,MIN,MERGE
  MERGE : called by BITSORT,MERGE

Called Intrinsics :
===================

SUBROUTINE ALLOCATE --
  ALLOCATE : called by BSORT
SUBROUTINE CMF_RANDOM --
  CMF_RANDOM : called by BSORT
SUBROUTINE DEALLOCATE --
  DEALLOCATE : called by BSORT
FUNCTION MAX --
  MAX : called by MERGE
FUNCTION MIN --
  MIN : called by MERGE

Called Externals :
==================
```

Figure 4: Call Graph Information

5.9 Adapt

The command Adapt is responsible for the generation of the new parallel program and the generation of a corresponding Makefile. This command has the same functionality as the use of Adaptor in the batch version.

In the interactive version the generated sources for the host and nodes will be shown in a window if the translation was successful (see figure 5).

If there have been any errors in the translation during a phase of the translation, the corresponding protocol file of the phase will be displayed in a new generated window that has the same functionality as the window for displaying the call graph information.

Quit | Print host.f | Print node.f

host.f

```
      SUBROUTINE HOSTMODULE ()
      INTEGER*4 N
      INTEGER*4 S
      INTEGER*4 K
      LOGICAL*1 A_SC1
      INTEGER*4 I_1
      PRINT *,'Input n for counting primes in range 2 to n : '
      READ *, N
      call dalib_broadcast (N,4,0)
      call dalib_clear_timer (1)
      call dalib_start_timer (1)
      K = 2
      DO WHILE (K*K .le. N)
        K = K+1
        call dalib_node_get (A_SC1,A_SC1,1,N,K)
        DO WHILE ( .not. A_SC1)
          K = K+1
          call dalib_node_get (A_SC1,A_SC1,1,N,K)
        END DO
      END DO
      S = 0
      call dalib_reduction (S,7)
```

node.f

```
      call dalib_start_timer (1)
cdir$ ivdep
      DO I_1=A_LOW,A_HIGH
         A(I_1) = .TRUE.
      END DO
      IF (dalib_have_i(N,1)) THEN
         A(1) = .FALSE.
      END IF
      K = 2
      DO WHILE (K*K .le. N)
        call dalib_local_range (N,K*K,N,K,A_START,A_STOP,A_INC)
cdir$ ivdep
        DO I_1=A_START,A_STOP,A_INC
           A(I_1) = .FALSE.
        END DO
        K = K+1
        call dalib_node_get (A_SC1,A(K),1,N,K)
        DO WHILE ( .not. A_SC1)
          K = K+1
          call dalib_node_get (A_SC1,A(K),1,N,K)
        END DO
      END DO
      S = 0
cdir$ ivdep
      DO I_1=A_LOW,A_HIGH
         IF (A(I_1)) THEN
```

Figure 5: The generated host and node program

5.10 Check

The command Check is only for test purposes. It checks the abstract tree for correct typing. The checking can be done after parsing and semantic analysis.

5.11 Write

Some users might be interested in the internal representation of the abstract syntax tree that stands for the program. When selecting Write the abstract syntax tree of the whole program is written to the file **test.out**. This file is not displayed in a new window. The command can be executed after parsing and semantic analysis.

It should be noticed that this file becomes very large for programs with many source lines. The abstract syntax tree of a single unit can also be displayed interactively by selecting the corresponding command in the unit area as it is explained in section 6.1.2.

5.12 Unparse

The creation of a new source program from an abstract syntax tree is called unparsing. With the command Unparse the current abstract syntax tree is unparsed and written to the file unparse.f that is also displayed in a new window.

5.13 Options

If the command Options is selected, a new menu is popped up where the user can choose the target language, the target machine, the programming model, and where he can choose between static and dynamic arrays.

All these options have only an effect for the translation which is called by the command Adapt.

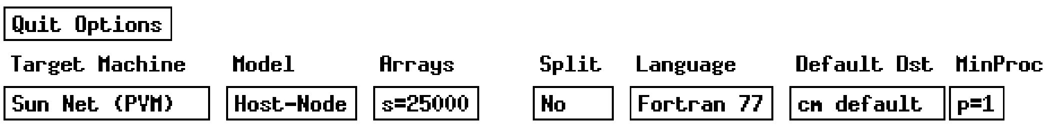


Figure 6: The Options Menu

The effect of the options is the same as if they are used in the batch translation (see section 3).

If an option requires a value (static size or minimal number of processors), this value can be defined within a new window. The value is accepted by selecting Accept or with the `RETURN`-key.

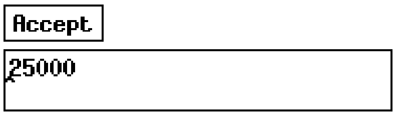


Figure 7: Interactive Input of a Value

6 Interactive Analysis of Units and Variables

After the semantic analysis has finished, it is possible to get information about units and variables of the source program.

6.1 Unit Menu

In the *Unit Area* the user can select a unit. After this selection the variables within the unit are listed in the *Variable Area* and the corresponding declaration of the unit is highlighted in the *Edit Area*.

The unit menu is pulled up when pushing the right mouse button in the unit area (see figure 8).

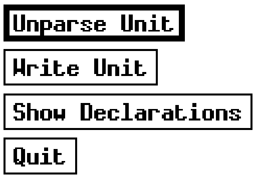


Figure 8: The Unit Menu

6.1.1 Unparse Unit

With this command the selected unit is unparsed and written to the file **unparse.f**. This file is afterwards displayed in a new window.

6.1.2 Write Unit

With this command the abstract syntax tree of the highlighted unit is written in ASCII format to the file `test.out`. This file is afterwards displayed in a new window.

6.1.3 Show Declarations

This command generates an ASCII file `test.sem` that contains information about all the variables within the selected unit. This file will be displayed in a new generated window.

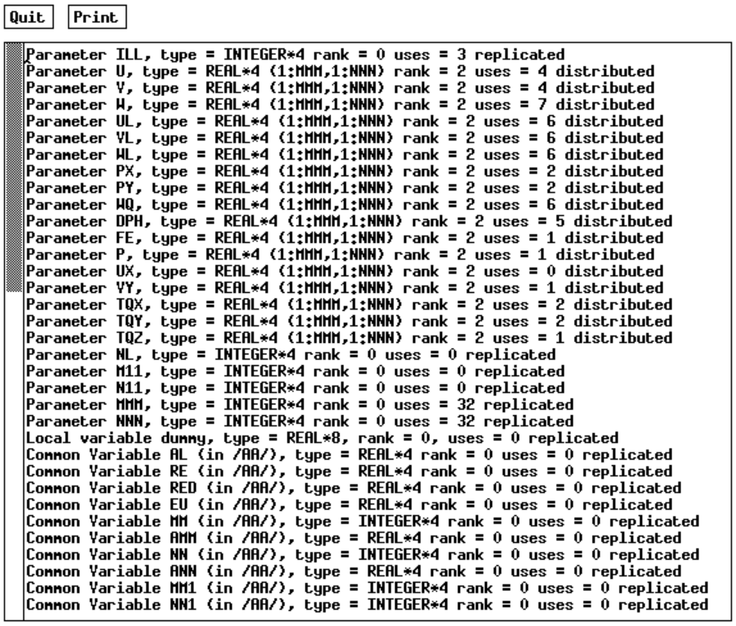


Figure 9: Information about the declarations of a unit

6.2 Var Menu

When selecting a variable in the variable area the corresponding declaration is displayed and the declaration is highlighted in the unit area. At the same time information about the variable is printed.

7 Compiling and Linking

When the translation of the data parallel program to a parallel program with explicit message passing is completed, the new programs have to be compiled and linked. These programs are

- **host.f** and **node.f** if the Host-Node model is used
- **cube.f** if the only node model is used

- **node1.f** if a sequential program has been generated

In some situations the length of source lines can be longer than 72 or 132 characters so that the compiler will fail. A typical error message is the following one:

```
Line 162  Error message # 1139
  Perhaps missing RPARENT before ENDOFST
   --  [unexpected end of statement]
```

If this is the case, the longer lines can be striped automatically with the following command of Adaptor:

```
fstrip cube.f 72    ! maximal 72 characters in one line
fstrip node1.f      ! default length is 132
```

For compiling and linking a **Makefile** is automatically generated with the translation. The 'Makefile' is different for the different parallel machines. So it should be possible to make the executables for the parallel program by simply typing 'make'.

If the Makefile does not work correctly, please read the 'Installation Guide of Adaptor' carefully, or make your own changes to the 'Makefile'. If a 'Makefile' is in the current directory, Adaptor will not generate a new one.

If the executables are no longer needed one can type

```
make clean
```

to delete all files that are no longer needed. Another command helps to delete all generated files by Adaptor:

```
adapt.clean
```

8 Running the parallel program

8.1 Start of the Processes

How the parallel programs are started is very machine-dependent.

For the Host-Node programming model, only the host process (host) is usually started. This process will automatically invoke the node processes (node).

For the Only-Node programming model the node program (cube) will be loaded on all node processors.

A single node program (node1) is started in the same way as other programs.

8.2 Number of Node Processes

The default number of node processes is usually the number of processors that has been reserved for running the parallel application. If such a reservation does not exist, there will be no default value.

The number of nodes can also be specified by giving an explicit argument or by setting the environment variable NP.

```
host 12          ! will start 12 node processes

setenv NP 12
host             ! will start 12 node processes
```

If there is no default value and no explicit specification for the number of processes, this number will be interactively asked for until a legal value has been given as an input. This is also done if there was an illegal explicit specification, e.g. if the number of node processes is bigger than the number of available nodes.

8.3 Using PVM

8.3.1 About PVM

The public domain software PVM [Sun90] is used for running the parallel program on a net of workstations, e.g. SUN 4 or IBM Risc. It can also be used to let different processes communicate via socket communication on a shared memory machine, e.g. the Alliant FX/2800, KSR 1 or SGI multiprocessor machines. It should be mentioned that this software also guarantees that Adaptor can be used for other kinds of workstations, but at the current state this has not been tested.

Adaptor supports PVM version 2.4 and 3.1. Refer to your installation manager to verify which version is used on your machine.

First experiences have shown that this environment can be used to test the parallel programs. But due to the high latency of communications when using workstations in many cases a good speed-up of the parallel program cannot be expected.

8.3.2 Running PVM Programs

Before starting the parallel program the PVM daemon has to be started. This daemon will run on all machines of the current configuration.

```
brasun 1 > pvm
pvm> add sprsun
1 successful
```

```
                  HOST     DTID
                sprsun    80000
pvm> add fourier
1 successful
                  HOST     DTID
               fourier    c0000
pvm> conf
3 hosts, 1 data format
                  HOST     DTID     ARCH   MTU SPEED
                brasun    40000     SUN4  4096     1
                sprsun    80000     SUN4  4096     1
               fourier    c0000     SUN4  4096     1
pvm> quit

pvmd still running.
brasun 2 >
```

Before starting the host program that loads the node program on all other workstations it should be guaranteed that the node program is accessible by all other workstations. This is usually done by a remote copy of the node program to all other workstations.

```
brasun 2 > rcp cube sprsun:pvm3/bin/SUN4
brasun 3 > rcp cube fourier:pvm3/bin/SUN4
```

8.4 Alliant FX/2800

Though the Alliant FX/2800 is a multiprocessor system with a shared memory, the parallel program will execute on this machine by running independent processes that communicate with each other (no automatic parallelization of the compiler is used).

The realization on the Alliant is done in such a way that the started process forks itself where the father process calls the host program and the child processes call the node program. The message passing between the processes is implemented on a shared memory region that will be created by the initial process.

If there is no host process, the first node process has to be started that forks itself in a similar way.

8.5 KSR 1

The KSR 1 is from the programmers point of view a shared-memory architecture like the Alliant FX/2800. The message passing is realized via a shared memory segment and by using semaphores. These Unix System V features are supported on the KSR machine.

8.6 Silicon Graphics

On the SGI parallel machine the same Unix System V features are provided as on the KSR machine. Both realizations of the DALIB are nearly identical.

8.7 iPSC/860

Before starting a parallel program on the iPSC, the user has to attach to a cube with the command **getcube**.

```
getcube -tn   (reservation of a cube with n processors)
```

When using a host process this process has to be started on the SRM (the current version does not support remote hosts), it will load the node processes automatically.

```
host   (will load node processes on all reserved nodes)
```

In the current version it is not possible to limit the number of node processes to a number smaller than the number of reserved nodes.

Without a host program the user has to load explicitly the node program on all nodes.

```
load cube;  waitcube
```

After the termination of the parallel program the cube should be released by using the command **relcube**.

```
relcube
```

8.8 Meiko CS

Executing the parallel program on a Meiko CS1 or CS2 is similar to running it on an iPSC. Please look in the manuals to find out the specific commands for running programs on this machine.

8.9 Parsytec GC

For the Parsytec system the generated Fortran sources must be stripped to a length of 72 characters.

```
fstrip [ host.f | node.f | cube.f | node1.f ] 72
```

For the Parsytec GC system one executable (host.px or cube.px) is generated that has to be loaded on all nodes with the run command.

```
run -s0 -g1 4 4 host   (will use 16 processors)
```

If there is a host process, one processor will execute the host program, all other processors the node program. If the executable is loaded on 16 processors only 15 processors will execute the node program.

Without a host program, only one executable (cube.px) is generated that has to be loaded on all nodes with the run command, too. In this case all processors will execute the node program, where node 1 is also responsible for I/O.

```
run -c2 4 4 cube   (will use 16 processors)
```

For the current version there are some strong restrictions concerning the fact that sometimes messages cannot be longer than one kByte.

8.10 CM-5

The number of actual used processors can be smaller than the number of available processors.

9 Performance Visualization

For some machines it is possible to collect run time data that can be visualized and animated. Currently, this works only for the Alliant machine.

If the host or cube program is started with the flag '-t', run time data will be collected. Another possibility is to set the environment variable **TRACE**.

```
setenv TRACE on
setenv TRACE ON
setenv TRACE 1
```

The following events are used to generate an entry for the tracefile:

- sending a message
- waiting for a message
- receiving a message

The collected trace data is sorted and one tracefile (with the name **tracefile**) is generated. The data can be viewed and animated with the public domain software ParaGraph [HE91]. It allows the resulting trace data to be replayed pictorially and provides a dynamic depiction of the behavior of the parallel program. There are up to 26 distinct visual perspectives from which to view the same performance data.

```
host -t 4 ...
....
tracefile created

paragraph tracefile
```

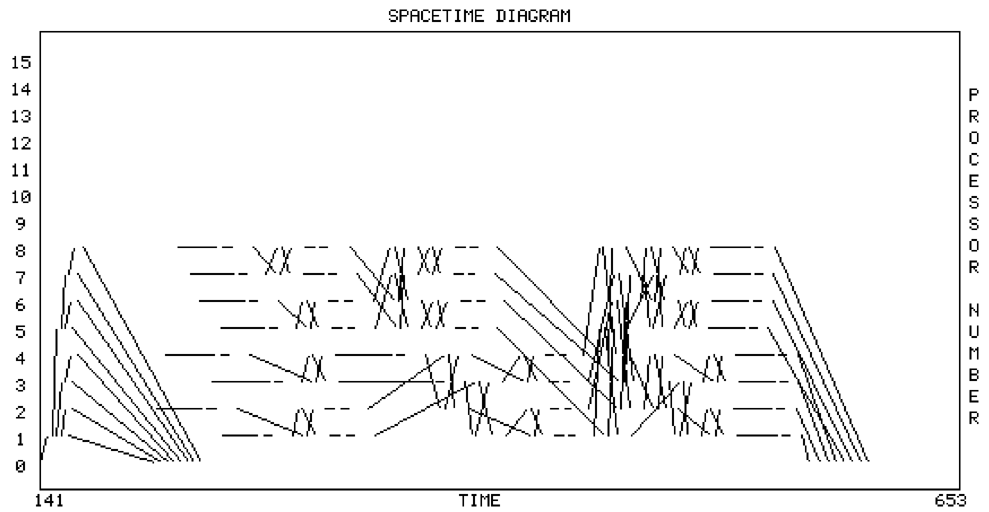


Figure 10: Example of Paragraph: Spacetime

10 Problems

If any problems occur while using Adaptor, please mail these problems to the author. It will be tried to fix these problems and to remove them immediately.

The author would also be very grateful if there were proposals for improvements. New applications that have been translated successfully with Adaptor are appreciated.

	Aggregate	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7	node 8
Percent Processor Busy	28	57	55	44	59	49	47	48	48	47
Percent Processor Ovhd	0	0	0	0	0	0	0	0	0	0
Percent Processor Idle	72	43	45	56	41	51	53	52	52	53
Number Msgs Sent	184	1	25	22	23	22	24	22	23	22
Total Bytes Sent	1440	4	188	176	180	176	184	176	180	176
Number Msgs Rcvd	184	16	21	21	21	21	21	21	21	21
Total Bytes Rcvd	1440	128	164	164	164	164	164	164	164	164
Max Queue Size (count)	8	8	2	2	2	2	2	2	2	1
Max Queue Size (bytes)	64	64	16	16	16	16	16	16	16	8
Max Msg Sent (bytes)	8	4	8	8	8	8	8	8	8	8
Max Msg Rcvd (bytes)	8	8	8	8	8	8	8	8	8	8

Figure 11: Example of Paragraph: Statistics

References

[Bra93a] T. Brandes. ADAPTOR Installation Guide (Version 1.0). Internal Report ADAPTOR-1, GMD, June 1993.

[Bra93b] T. Brandes. ADAPTOR Language Reference Manual (Version 1.0). Internal Report ADAPTOR-3, GMD, June 1993.

[HE91] M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[NO90] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual.* Nutshell Handbooks, Sebastopol, CA, 1990.

[O'R90] T. O'Reilly. *X Toolkit Intrinsics Reference Manual.* Nutshell Handbooks, Sebastopol, CA, 1990.

[Sun90] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 3(10), December 1990.