HPF+

Functional description of the performance

M. Calzarossa, L. Massari, A. Merlo, D. Tessera
Dipartimento di Informatica e Sistemistica
University of Pavia
via Ferrata 1, 27100 Pavia, Italy

Deliverable:	D2.3b
Date:	March 1998
Status:	draft
Confidentiality:	public
Responsible:	University of Pavia

Revised:

The **HPF+** Consortium:
AVL GesmbH - Austria
European Centre for Medium-Range Weather Forecasts (ECMWF) - England
Engineering System International SA (ESI) - France
NA Software Ltd (NAS) - England
University of Pavia - Italy
University of Vienna - Austria

Contents

1 Introduction

This report describes the extensions of Medea aimed at the functional description of the performance of the HPF+ kernels. The objective of such extensions is to correlate the obtained performance with specific regions of source code.
A functional description of the performance is only possible if there is a tight integration between the VFC compiler and the Medea tool. Starting from high level HPF+ directives of the source code, the VFC compiler generates an explicit message passing parallel code, to be executed on a parallel system. Application developers have very little control on this parallel code. Relationships between performance and the HPF+ source code are very helpful for application developers in order to understand the influence of the parallelization strategies on the performance.
Medea has been enhanced to address all these issues. New features have been added to the filtering module. New types of analyses, together with new visualization facilities, have been integrated within Medea.
Medea provides a quantitative description of the performance, based on measurements [1]. From tracefiles, performance figures, such as, profiles, metrics and parameters are derived. Moreover, Medea has been enhanced in order to evaluate the performance of the kernel benchmarks from a qualitative viewpoint. The performance of specific code regions, such as, sequential DO loops, subroutines, `INDEPENDENT` loops, can be analyzed, by deriving for each of them various performance figures. Furthermore, the contributions of the various phases of the parallelization strategies applied by the compiler can be recognized within the total execution time of `INDEPENDENT` loops and subroutines. The communication times are seen as the sum of times taken by the different communication protocols.
All these kinds of analysis provide both application and compiler developers with source–level information about the performance achieved by the kernel.

The report is organized as follows. In Section 2, the integration between VFC and Medea, and the new filtering module are presented. In Section 3, the functional description of the performance is presented by showing various types of results. The appendices include a few more technical details.

2 Integration of Medea and VFC

As a preliminary phase towards the functional description of the performance, it was necessary to identify the information to be collected by the measurement system (SIS). Then, we analyzed the new functionalities to be integrated into the filtering module of Medea, in order to process this collected information. Information, used for deriving a functional description of the performance of the code, are related, for example, to timings of the five phases of the inspector/executor strategy, i.e., work distributor, inspector, gather, executor and scatter [2]. In what follows, the structure of the tracefiles and the formalism used for defining the stored information are described.

2.1 The SIS measurement system

In order to support performance analysis of HPF+ codes, VFC provides SIS, a system for automatic program instrumentation and monitoring.
SIS is able to instrument, via command–line compiler options, various code regions, such as, subroutines, `INDEPENDENT` loops and sequential DO loops. SIS analyzes the HPF+ source code and collects compile–time information regarding each code region to be measured. Then, SIS instruments the various code regions, while they are parallelized by VFC, as well as the additional pieces of code inserted by the compiler into the generated code as a consequence of the applied parallelization strategies. For example, in the case of `INDEPENDENT` loops, each individual phase of the inspector/executor strategy is instrumented. In the case of parallelization of a subroutine call, SIS instruments the call itself and the pieces of code inserted by the compiler before and after the call, i.e., its head and tail.
SIS is also able to instrument all the communication activities and to identify the various communication protocols.
The instrumentation of the code regions and/or of the additional pieces of code is achieved by marking the begin and the end of each code region by means of measurement points. During the execution of the code, timestamps and identifiers for each measurement point are collected.

At the end of the execution of the generated parallel code, a tracefile containing timings for each measurement point and for the various communication activities is produced.

We have adopted the following naming convention for the identifiers associated to each measured code region:

`<source>.<linenumber>.<label>`

where:

- `source` is the name of the source file containing the code region
- `linenumber` is line number of the HPF+ code where the code region starts
- `label` is an identifier of the instrumented region, which can take one of the following values:
  - `PR`: whole kernel
  - `II`: inspector
  - `IW`: work distributor
  - `IG`: gather
  - `IE`: executor
  - `IS`: scatter
  - `SH`: subroutine head
  - `SE`: subroutine call
  - `ST`: subroutine tail

– `LP`: sequential DO loop.

These identifiers, together with the timings contained into the tracefile, are used by Medea for a source–level performance analysis of the HPF+ code. Each code region and its additional pieces of code can be immediately identified and analyzed. Note that the execution times of these additional pieces of code represent the costs of the parallelization, usually referred to as the overhead associated to the parallelization strategies.
In Appendix B, an example of a portion of a tracefile produced after the execution of an HPF+ kernel is shown. The first number of each line identifies the type of information the line contains. Lines beginning with `-13` contain the association between the identifier of code regions, e.g., `KFORCE.32.IE`, and the identifiers of the corresponding measurement point, e.g., `1072`, `1073`.

2.2 The filtering module of Medea

The filtering module of Medea has been enhanced in order to correlate the information contained into tracefiles and the performance of the HPF+ kernels at the source code level. Performance analysis covers both the whole kernel and specific code regions (e.g., `INDEPENDENT` loops, subroutines). For each of the instrumented code regions, the filter provides a compact description of the performance, in terms of execution time, number of occurrences and communication time. As already pointed out, these kinds of results have been made possible by the tight integration between the compiler, the measurement system and Medea.

The new features provided by the filtering module of Medea are aimed at outlining the performance achieved by code regions as a function of the specified data distributions and of the parallelization strategies applied by the VFC.

The filter module computes the overhead associated to the parallelization strategies adopted by the compiler, as well as performance figures of the source code itself.
For example, for `INDEPENDENT` loops, five phases have been identified: work distribution, inspector, data gathering, execution and results scattering. The filter module derives performance figures for all these phases, as well as for the whole loop. As a consequence, the execution time of an `INDEPENDENT` loop is seen as the sum of the times attributed to the various phases.

Performance results are logically grouped together according to the type of code region they refer to, i.e., loops, subroutines and `INDEPENDENT` loops. The naming convention adopted by the filtering module of Medea follows SIS namings.

3 Functional description of the performance

As already pointed out, performance evaluation can be approached from two different viewpoints, that is, a quantitative and a qualitative viewpoint. The former provides detailed

information about the kernel performance, in that parameters, such as, execution, communication, and per–protocol times, are computed for the various code regions. These values are then shown in a tabular form by means of the visualization facilities of Medea, and can be used as input for the clustering module.
The qualitative viewpoint gives a logical description of the performance, in that, the derived parameters are "reinterpreted" in order to obtain meaningful representations of the achieved performance able to highlight the impact of the compiler in the parallelization process. In this section, we present the main characteristics of the `Functional` module of Medea.

This module, which is activated from the main window of Medea (see Fig. 1), deals with the profiling of the kernels. Profiling provides a functional, i.e., qualitative, description of the quantitative performance figures computed by the `Filtering` module.

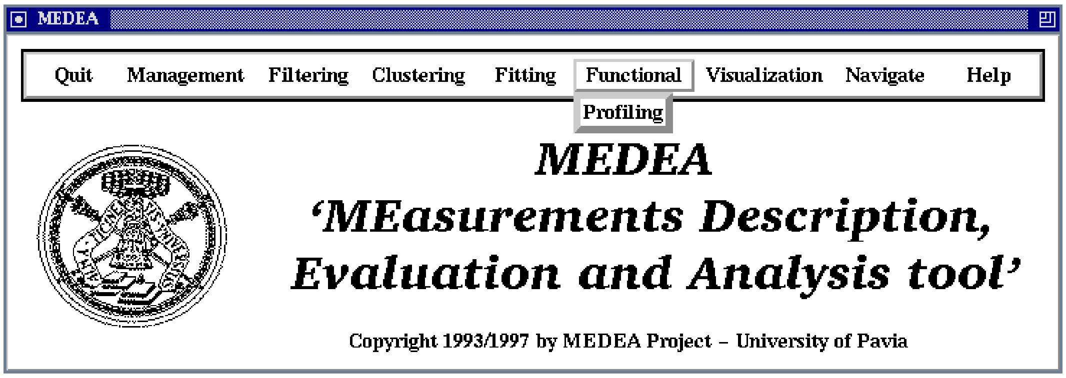


Figure 1: Main window of Medea.

In the following sections, we describe the `Profiling` module by showing a few results, derived by analyzing the tracefiles collected during two executions of the NEC benchmark kernel Version 2. The traces refer to runs with 8 and 32 processors, respectively. These traces contain the events related to subroutines, sequential DO loops, and `INDEPENDENT` loops.

3.1 The Profiling module

When this module is activated, qualitative descriptions of the performance achieved by the various code regions can be visualized. The main window of this module is shown in Fig. 2. This window is logically subdivided into a "service" area (containing `Quit`, `Analyses` and `Help buttons`) and four main working areas, namely, `Program Profiling`, `Communication Analysis`, `Component Profiling`, and `Source Code`. This subdivision reflects the different types of information that can be extracted by processing the tracefiles.

In our examples, the two tracefiles have been processed by means of the automatic run mode (see Appendix A) provided by Medea [1]. As a result, three different sets of performance results have been automatically obtained, one for each tracefile (namely, `nec-kernel_08p` and `nec-kernel_32p`) and a `global` one, which refers to the results of an analysis of both tracefiles.

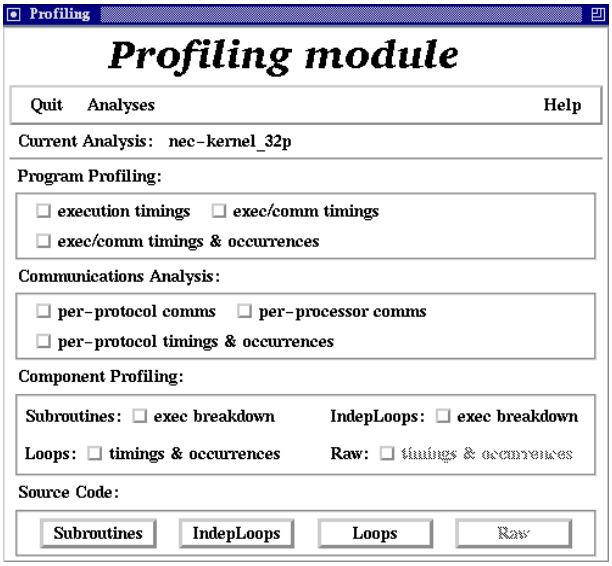


Figure 2: Main window of the `Profiling` module.

A specific set of results can be selected within the main window by means of the `Analyses` button, as shown in Fig. 3. In what follows, the different functionalities of the `Profiling` module of Medea are presented.

3.1.1 Program profiling

The results of the `global` analysis provide information concerning only the execution and communication times of the analyzed tracefiles. Comparisons among the overall behavior of a kernel executed with a different number of allocated processors can be easily addressed.
The `execution timings` button within the `Program Profiling` working area pops up a diagram displaying, by means of bar charts, the execution times of the analyzed runs (see Fig. 4). This type of visualization reveals very useful when analyzing multiple tracefiles to get a quick glimpse at the performance that can be achieved by varying the number of processors. The scalability of the kernel is also shown.

When we are interested to the results of the analysis of a single run of the kernel, i.e., a single tracefile, the `exec/comm timings` button pops up a diagram with more details on its behavior (see Fig. 5). The execution and communication times are visualized in order to address the computation/communication tradeoffs. Moreover, the contributions to the overall communication time due to the transmit, receive and collective operations are presented.
Note that all the times shown in the various diagrams are expressed in seconds.

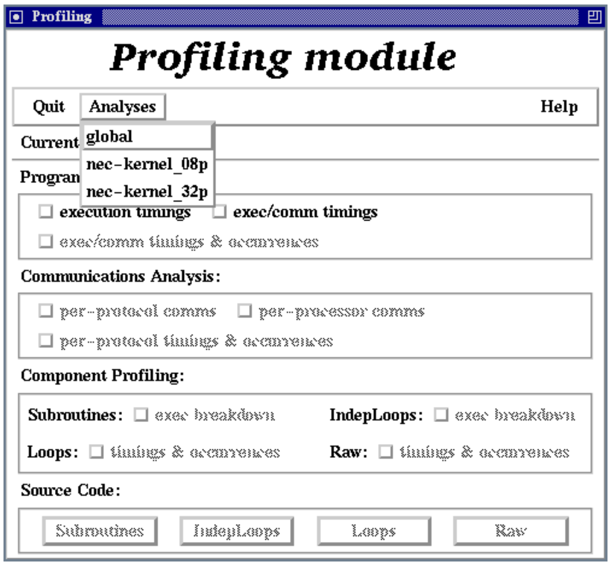


Figure 3: Selection of the results corresponding to the `global` analysis.

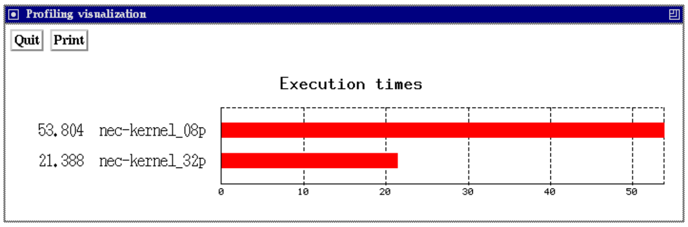


Figure 4: Execution times (expressed in seconds) of runs of the kernel with 8 and 32 allocated processors, respectively.

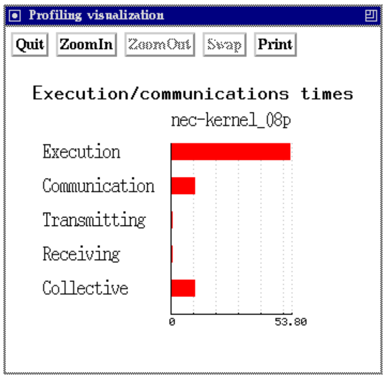


Figure 5: Execution and communication times (expressed in seconds) of a run of the kernel.

Moreover, the `exec/comm timings & occurrences` button displays the execution, communication, transmit, receive, and collective times together with the corresponding average

numbers of occurrences (see Fig. 6). The average is computed with respect to the number of allocated processors.
As an example, for the kernel run with 8 processors, the average number of collective communications is equal to 3628, and the time spent by each processor while performing all these operations is equal to 10.782 seconds. As a matter of fact, this time accounts for more than 99% of the total communication time.

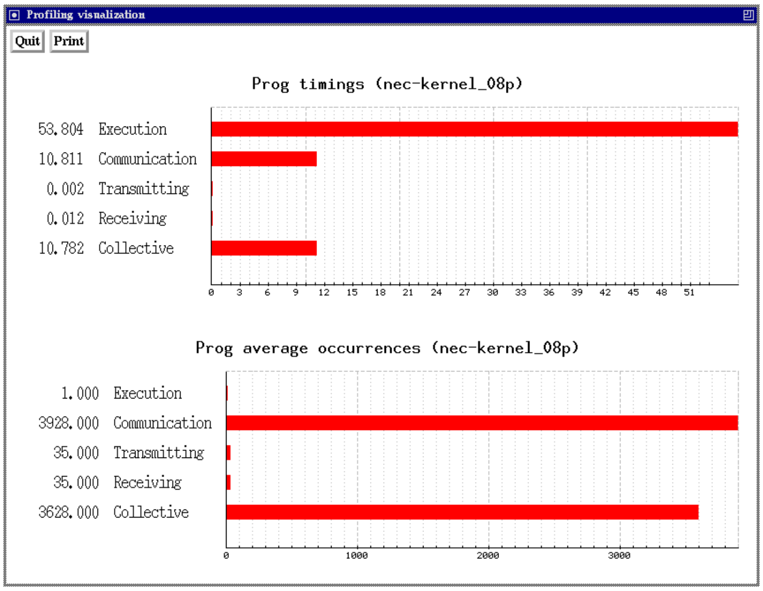


Figure 6: Execution and communication times (expressed in seconds) together with their corresponding average number of occurrences.

3.1.2 Communication analysis

The communication activities of the kernel are analyzed both on a per–protocol and a per–processor basis.

The `per-protocol comms` button pops up a window displaying the total communication time broken down into the contributions due to the different MPI communication protocols (see Fig. 7). In this figure, the communication time corresponding to the execution of the NEC kernel with eight processors is split according to the eight different protocols used. Each protocol is identified by a color. A legenda appears at the right hand side area of the window.

In order to ease the interpretation of these results, a few facilities have been added to this type of diagrams. By clicking on the colored square associated to each label, the corresponding slice is placed at the beginning of the bar, i.e., the bar starts with the selected color, and the remaining ones are shifted accordingly. Furthermore, since the communication time can be dominated by a few protocols, it may happen that even this operation does not succeed in displaying the time spent by a specific protocol when it is very small. In such a case, the

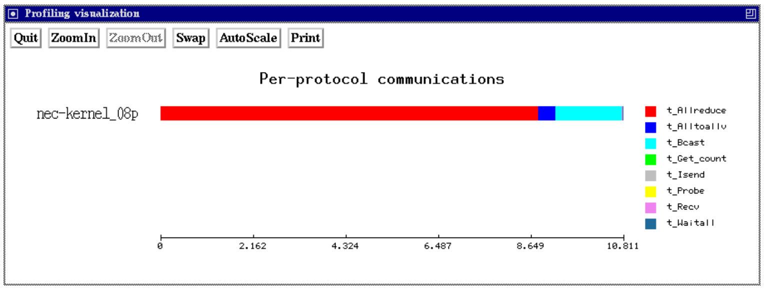


Figure 7: Per protocol communication analysis. Times are expressed in seconds.

`AutoScale` button at the top of the window automatically scales all the values in order to visualize even the smallest one (see Fig. 8).
In this figure, the `Get_count` protocol has been selected and its communication time has been visualized by means of the `AutoScale` facility. A scrollbar at the bottom of the window is associated to the `AutoScale` facility.

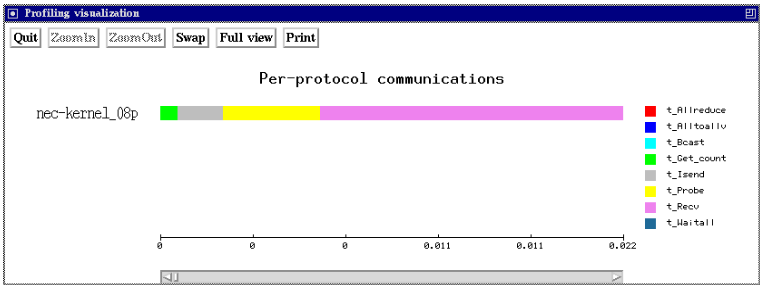


Figure 8: Autoscale applied to the per protocol communication diagram of Fig. 7.

Figure 9 shows the same information of Fig. 7 from a different viewpoint. Indeed, the `Swap` button at the top of the window has been used to present the times spent by each protocol.

The analysis of the communication time on a per–protocol basis can be further refined by means of the `per-protocol timings & occurrences` button. The overall communication time and the times of each protocol are displayed together with their average number of occurrences by means of bar charts (see Fig. 10). These diagrams reveal very useful, for example, when tradeoffs between the time spent within a specific protocol and the number of corresponding operations executed by each processor have to be addressed.

More details on the performance of the kernel with respect to the communication activities can be pointed out on a per–processor basis, i.e., by looking at the behavior of each allocated processor. The communication time of each processor can be broken down into the contributions due to the different communication protocols, as shown in Fig. 11. The communication times are scaled with respect to the highest value. In such a way, unbalanced conditions among the processors can be easily recognized.

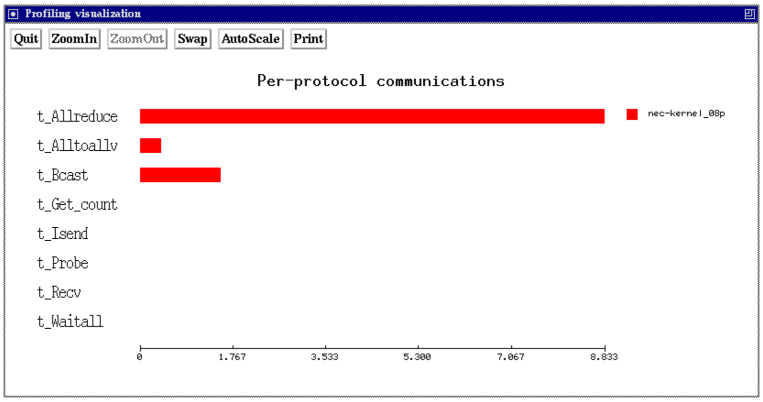


Figure 9: Swap facility applied to the per protocol communication diagram of Fig. 7.

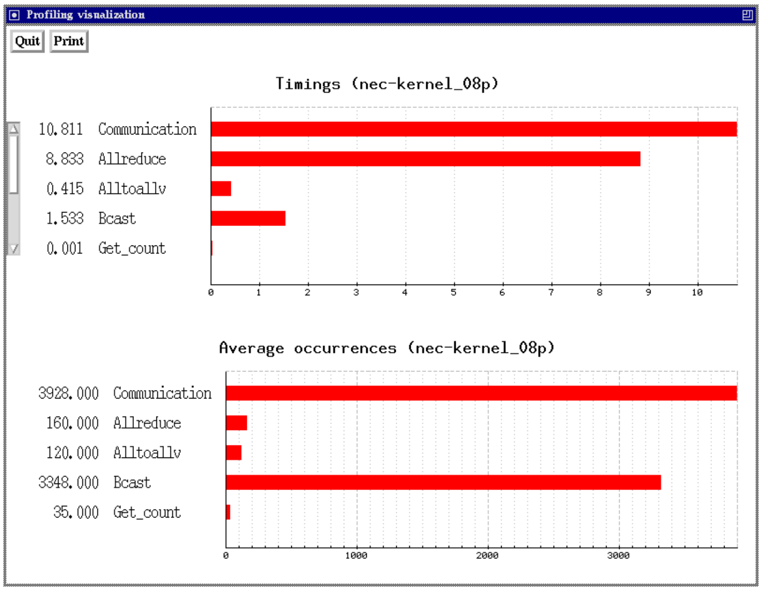


Figure 10: Overall and per–protocol communication times (expressed in seconds) and average number of occurrences.

The same visualization facilities described for the per–protocol analysis are provided for the per–processor analysis. For example, the analysis can be focused on a specific protocol by selecting the corresponding color and the autoscaling facility can be applied (see Fig. 12). Moreover, the same information can be presented under a reversed viewpoint, by means of the `Swap` button. Figure 13(a) shows a swapped diagram, where the time of each protocol is split according to the contribution given by each processor. Figure 13(b) shows a more detailed view by means of the `AutoScale` facility.

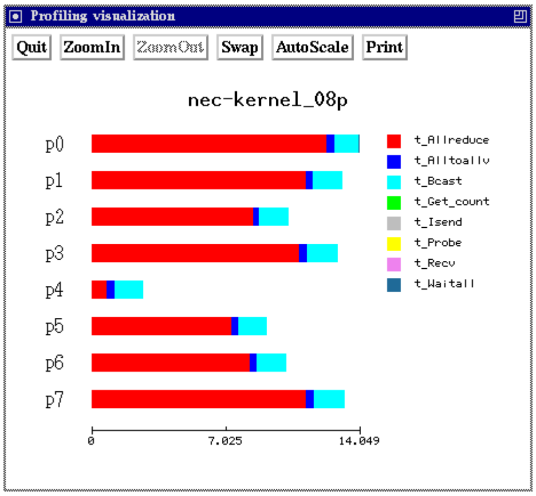
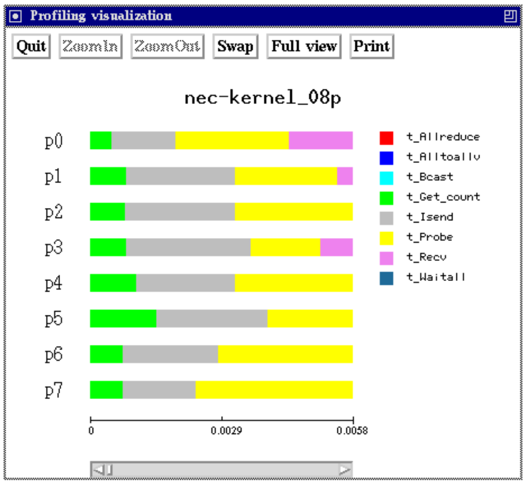


Figure 11: Breakdown of the communication time (expressed in seconds) of each processor into the times spent in the different communication protocols.

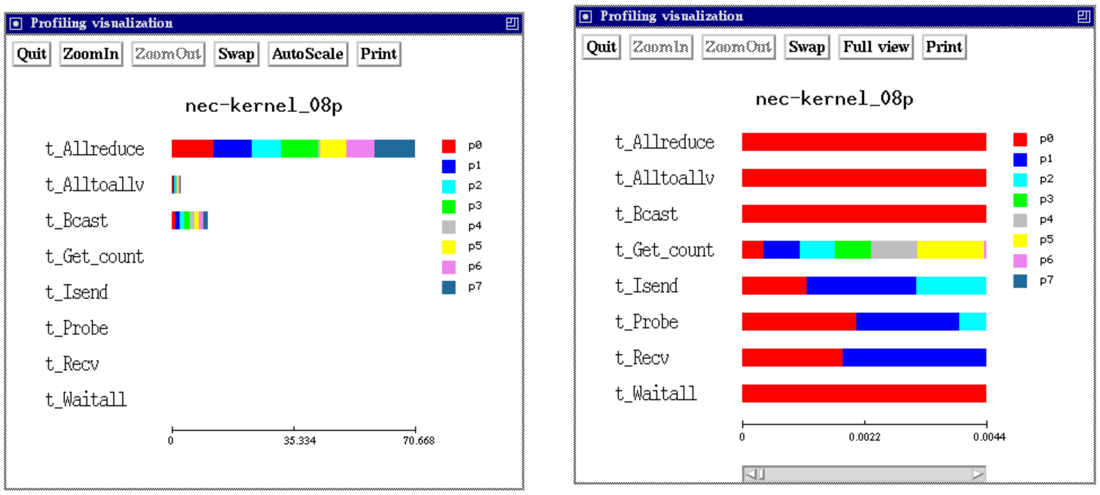


(b)

Figure 12: Autoscale of a per–processor communication diagram.

3.1.3 Component profiling

The profiling of the instrumented regions are addressed by means of the visualization facilities provided within the `Component Profiling` area of the `Profiling` module. When analyzing `INDEPENDENT` loops and subroutines, the profiling highlights the overhead due to the parallelization strategies adopted by the compiler for the specific code regions. As a consequence, the execution time of each subroutine or `INDEPENDENT` loop can be seen as the sum of times



(a) (b)

Figure 13: Swap (a) of the diagram of Fig. 11 followed by an autoscaling (b).

spent in the various phases, as explained in Section 2.2.

The `exec breakdown` button associated to `Subroutines` pops up a window where the overall execution time of each subroutine is visualized as the sum of the `Head overhead`, `Executor`, and `Tail overhead` contributions (see Fig. 14). Subroutines are labeled according to the name of the source file they belong to and their starting line number. As an example, the subroutine `UPDATE.86` belongs to the source file `update.hpf` and starts at line 86. Times are displayed in a decreasing order. If no overhead has been introduced by the compiler, the times of `Head` and `Tail` phases are zero. The swap facility is associated to this type of diagrams.

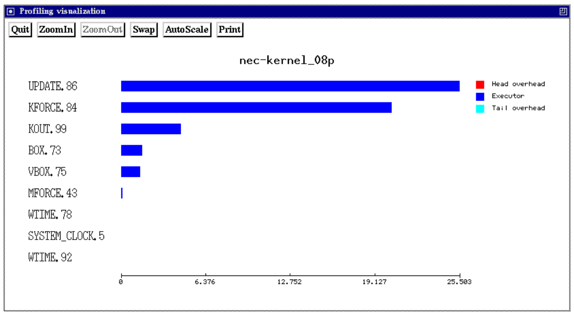


Figure 14: Breakdown of the total execution time (expressed in seconds) of the subroutines with respect to the overhead introduced by the compiler.

The same approach has been adopted to visualize the overhead introduced by the compiler while dealing with the parallelization of `INDEPENDENT` loops. In this case, the `exec breakdown` button pops up a window which displays, in a decreasing order, the execution times of the various `INDEPENDENT` loops. Each of these times is the sum of the times due to the five phases of the inspector/executor strategy, i.e., work distributor, inspector, gather, executor and scatter. The time of each phase is displayed in the diagram with a color (see Fig. 15).

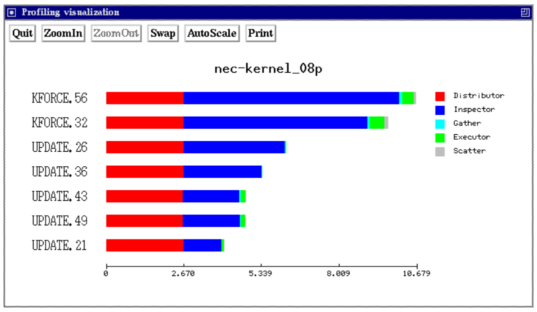
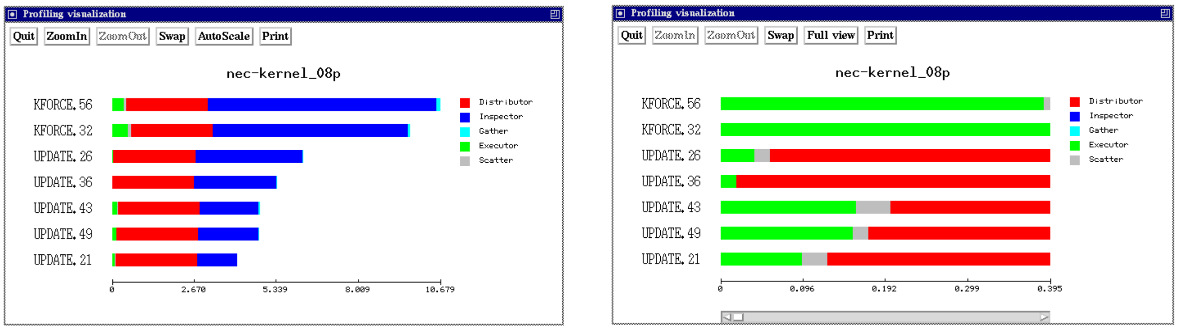


Figure 15: Breakdown of the total execution time (expressed in seconds) of the `INDEPENDENT` loops.

`INDEPENDENT` loops are labeled according to the same formalism adopted for subroutines. Comparisons among the times spent, for example, in the `Executor` phase across different `INDEPENDENT` loops can be easily addressed by selecting the color identifier associated to the legenda in the right hand area of the window (see Fig. 16(a)). The `AutoScale` button allows the visualization of the times even though they are very small, as shown in Fig. 16(b).



(a) (b)

Figure 16: Highlight of the `Executor` phase (a) and AutoScale (b).

More detailed diagrams are associated to `INDEPENDENT` loops and subroutines profiling. By clicking the right mouse button on a component label (e.g., `KFORCE.32` in Fig. 15) diagrams containing timings and occurrences are displayed. In the case of `INDEPENDENT` loops, the times refer to the overall execution, communication, distributor, inspector, gather, executor and scatter times spent by the component. Similarly, the average numbers of occurrences refer to the various phases. For example, Figure 17 shows the diagrams corresponding to the `KFORCE.32 INDEPENDENT` loop.
The same type of diagrams are provided for the subroutines.

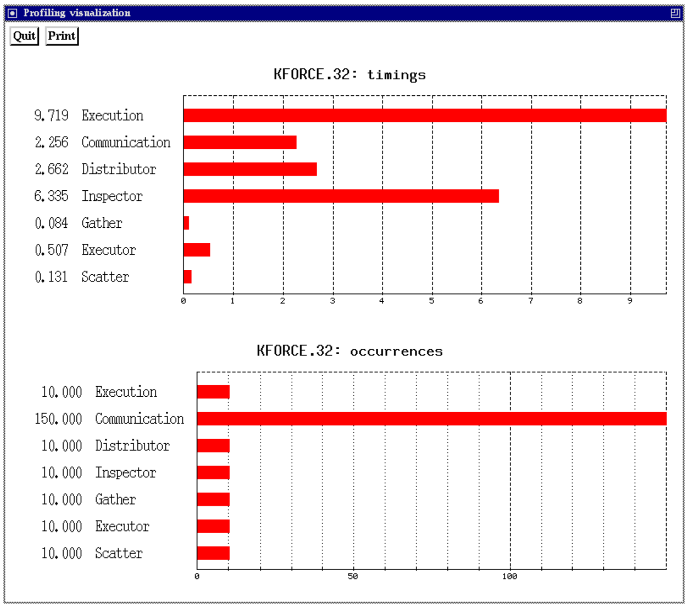


Figure 17: Timings and number of occurrences of an `INDEPENDENT` loop.

The instrumentation of sequential DO loops and of arbitrary (i.e., raw) components of the code produces only timings and occurrences information. These results are displayed by activating the `timings & occurrences` buttons corresponding to `Loops` and `Raw`, respectively. Figures 18 shows the execution times and the number of occurrences of the sequential loops. Note that the vertical scrollbar allows the user to scroll through the various loops.

The naming convention adopted to label the sequential loops follows the subroutine and `INDEPENDENT` loop naming. This convention differs for raw events. Indeed, in such a case, the name of the tracefile containing the event associated to an arbitrary code segment is also part of its identifier. Furthermore, no logical interpretation is applied to the raw events.

3.2 Source code

The adopted naming convention allows a link back to the source code. Indeed, the label associated to a particular diagram identifies the type of code region and its position within an HPF+ source file.

The source code of each instrumented code region is visualized by clicking the corresponding label. A window containing the HPF+ source code is popped up, and the identified code region is highlighted. As an example, by clicking on the `KFORCE.32` label in Fig. 15, the visualization of the source code of the corresponding `INDEPENDENT` loop is obtained (see Fig. 19).

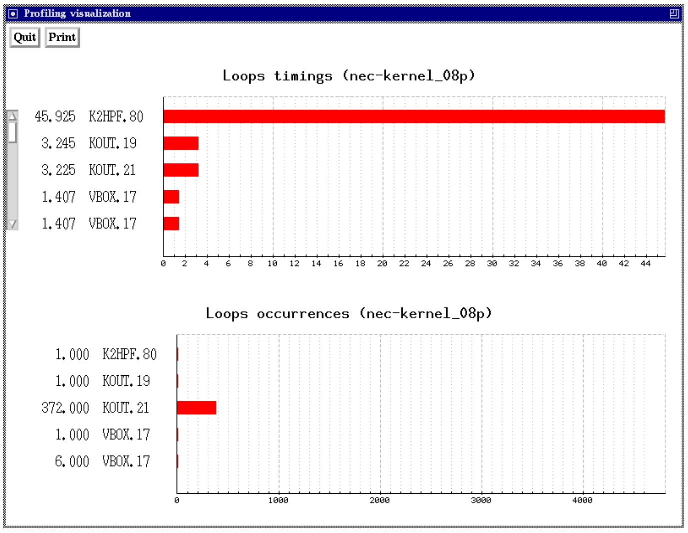


Figure 18: Execution times (expressed in seconds) and number of occurrences of the sequential loops.

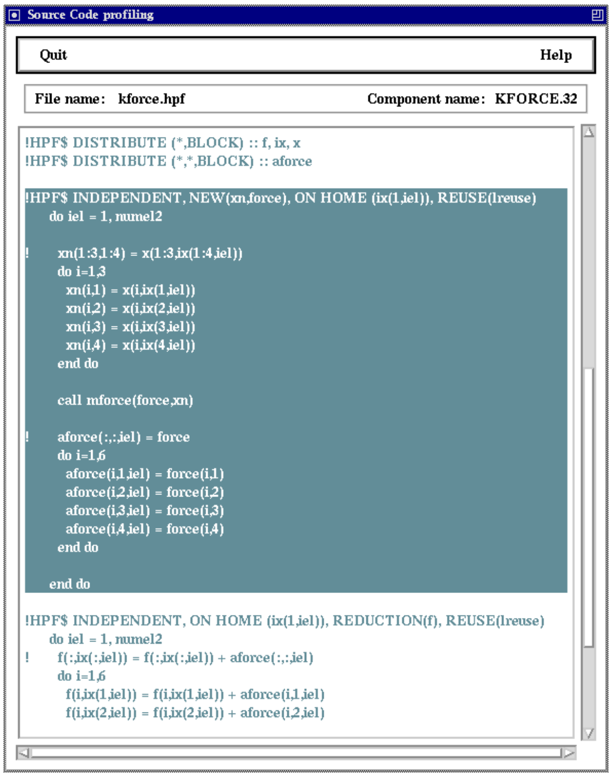


Figure 19: Source code visualization.

The visualization of the source codes is also provided in the `Source Code` area at the bottom of the main window of the `Profiling` module. Buttons corresponding to the identified region are made sensitive.
After the selection of a specific component, the list of code regions of the corresponding type is visualized. Each identifier consists of two components, namely, the name of the source file containing the component itself, and the line number within the source file. The source code of the loop shown in Fig. 19 is obtained, as shown in Fig. 20, by selecting the label `KFORCE.32`. The `INDEPENDENT` loop starting at line 32 of the file `kforce.hpf` will then be displayed.
Note that the directory containing the source files has to be specified as part of the Medea command line options (see Appendix A).

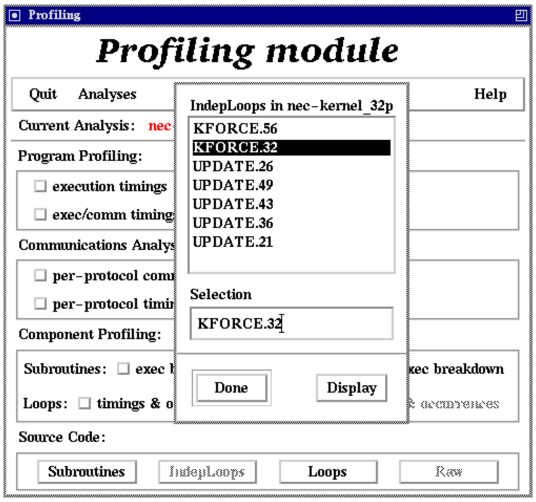


Figure 20: Component selection.

References

[1] M. Calzarossa, L. Massari, A. Merlo, and D. Tessera. Extension of MEDEA for the performance analysis of HPF+ benchmarks. Document HPF+ D2.3a, University of Pavia, March 1997.

[2] S. Benkner. Extension of VFCS for the Compilation of Project Benchmarks - Version 2. Document HPF+ D2.2b, Institute for Software Technology and Parallel Systems, University of Vienna, July 1997.

A Command line options

The following command line options are recognized by Medea.

`-session <session_name>`	This option is used to specify a session name. If the session does not already exist, it is created. Otherwise, the session with the specified name is restored and all the previous results are made available for visualization or further refinements by means of new analyses. This option can be used in combination with all the following ones.
`-trace <program>_<processors>`	This option is used to identify specific tracefiles to be automatically analyzed by Medea. Whenever invoked, this option forces Medea to scan its environment and to look for an MPICH tracefile and a VFC tracefile that match not only the `<program>` part but also the `<processors>` one. If the `-n` option is not specified, Medea requires a confirmation of the list of identified tracefiles before starting the automatic run.
`-mtrace <program>`	This option is used to identify the set of multiple tracefiles to be automatically analyzed by Medea. Whenever invoked, this option forces Medea to scan its environment and to look for all the tracefiles that match the specified `<program>` name. If the `-n` option is not specified, Medea requires a confirmation of the list of identified tracefiles before starting the automatic run.
`-n`	With this option, Medea disables the confirmations associated with the `-mtrace` and `-trace` options and provides only appropriate error/warning messages upon failures/mismatches detected after the automatic run has started.

-seq <sequential_time>	This option is used to specify the execution time of a run of the kernel with one processor. This value, which overwrites the times that could be derived from traces referring to single processor runs, is used for computing speedup, efficiency, and efficacy.
-quit	This option prevents the graphical interface of Medea to be displayed upon completion of an automatic run. If this option is not specified, at the end of the analysis performed in the automatic run mode the interface is popped up with the current session already active for visualizing the computed results.
-code <directory_name>	This option is used to specify the name of the directory which contains the source files to be used by the `Functional` module of Medea. The `<directory_name>` can be either an absolute path name or a path name relative to the root directory of the Medea environment. In both cases, the existence of this directory is checked. All the tracefiles parsed during the automatic analysis are linked to the sources in this directory.
-version	This option shows the version of Medea.
-help	This option shows the different command line options recognized by Medea.

B Example of a tracefile

```
-1 0 0 0 0 0 Me
-2 0 0 595168 0 0
-3 0 0 8 0 0
-4 0 0 1 0 0
-5 0 0 0 0 0
-6 0 0 0 0 9
-7 0 0 0 0 53803766
-8 0 0 1 0 0
-11 0 0 0 0 0
-13 0 1000 1001 0 0 white:black  BOX.23.LP
-13 0 1002 1003 0 0 white:black  BOX.24.LP
-13 0 1004 1005 0 0 white:black  BOX.30.LP
-13 0 1006 1007 0 0 white:black  BOX.36.LP
-13 0 1008 1009 0 0 white:black  BOX.42.LP
-13 0 1010 1011 0 0 white:black  BOX.50.LP
-13 0 1012 1013 0 0 white:black  BOX.51.LP
-13 0 1014 1015 0 0 white:black  K2HPF.1.PR
-13 0 1044 1045 0 0 white:black  BOX.73.SE
-13 0 1046 1047 0 0 white:black  VBOX.75.SE
-13 0 1048 1049 0 0 white:black  WTIME.78.SE
-13 0 1058 1059 0 0 white:black  K2HPF.80.LP
-13 0 1050 1051 0 0 white:black  KFORCE.84.SE
-13 0 1052 1053 0 0 white:black  UPDATE.86.SE
-13 0 1054 1055 0 0 white:black  WTIME.92.SE
-13 0 1056 1057 0 0 white:black  KOUT.99.SE
-13 0 1072 1073 0 0 white:black  KFORCE.32.IE
-13 0 1070 1071 0 0 white:black  KFORCE.32.IS
-13 0 1068 1069 0 0 white:black  KFORCE.32.IG
-13 0 1066 1067 0 0 white:black  KFORCE.32.II
-13 0 1064 1065 0 0 white:black  KFORCE.32.IW
-13 0 1086 1087 0 0 white:black  KFORCE.37.LP
-13 0 1084 1085 0 0 white:black  MFORCE.43.SE
-13 0 1088 1089 0 0 white:black  KFORCE.46.LP
-13 0 1082 1083 0 0 white:black  KFORCE.56.IE
-13 0 1080 1081 0 0 white:black  KFORCE.56.IS
-13 0 1078 1079 0 0 white:black  KFORCE.56.IG
-13 0 1076 1077 0 0 white:black  KFORCE.56.II
-13 0 1074 1075 0 0 white:black  KFORCE.56.IW
-13 0 1090 1091 0 0 white:black  KFORCE.59.LP
-13 0 1092 1093 0 0 white:black  KOUT.14.LP
```