REAPAR User Manual and Reference:

Automatic Parallelization of Irregular Recursive Programs

Stefan U. Hänßgen
haenssgen@ira.uka.de

Interner Bericht 08/98
Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
Karlsruhe, Germany

14-mar-98

Abstract

This report describes the REAPAR system (REcursive programs Automatically PARallelized) for automatic instrumentation, parallelization strategy selection and parallelization of recursive programs.

It presents the background required for understanding the system and explains how a user can automatically instrument a program for producing recursion profile information, how to derive a thread-parallel program from recursive ANSI C source code, and how parallelization strategies are selected automatically. Also, the report explains the options offered by the system and gives troubleshooting advice and workarounds for system limitations.

Supplemental chapters outline the structure of the system and describe the options in depth. They also detail the algorithms used for instrumenting code to generate recursion profile information, code insertion to allow for thread-parallel execution, and selection of parallelization strategies for both fine and coarse grained programs. Additionally, example of source code before and after transformation, examples for speedups of different parallelization strategies, and a summary of speedup and strategy selection results are given.

Contents

Chapter 1

Introduction

This chapter describes background information for the REAPAR system and gives an overview of the report. Users who want to start experimenting right away can directly skip to chapter 2 and later return here for more information.

1.1 Background, Goal, and Methods Used

REAPAR is a system that parallelizes recursive programs automatically.

Writing efficient parallel programs still is a difficult task, even after decades of research into the topic. For data-parallel programs with regular data access patterns, automatic parallelization has made substantial progress [Wol96], but irregular programs remain a challenge. An interesting subclass of irregular programs are those whose parallelism is implicit in multiple independent recursive calls that are started from a loop or statement sequence. Such procedures, often found in computations on graphs or trees, form the core of a number of real-life applications, e.g., Barnes Hut [BH86] galaxy simulation or algorithms used in computer graphics.

Of course, recursive programs can be brought into iterative form, but this often results in a less understandable program and, most importantly, will still not allow for efficient automatic parallelization if the computational structure is irregular.

The purpose of this technical report and manual is to show that automatic parallelization is practical for many recursive programs on Shared Memory Multiprocessor (SMP) machines. As a result, SMP parallelism can efficiently be used by programmers without knowledge of parallel programming.

For introducing parallelism, we replace each parallelizable recursive call by a call that possibly introduces a new thread for the current recursion step.

Additionally, recursion profile information is often useful to gain further insights into how a program works and which parallelization strategy performs best. For obtaining this information, the system automatically instruments the user program to collect and output profiles at runtime.

Together with the automatic selection of an appropriate parallelization strategy based on the recorded profiles, these components form a system called REAPAR (for "REcursive programs Automatically PARallelized" — reaping the fruits of potential parallelism).

1.2 Overview

The report is structured as follows:

1. **Introduction and Overview:** What you are reading at this very moment.
2. **Using the System:** User manual presenting the system's environment and requirements, describing the benchmarks, the system's components, their features, and giving examples of their usage.
3. **Advanced Features:** Explains the system's options and their use. Gives background information on possible limitations and describes how to overcome them.
4. **Recursion Analysis:** In-depth description of how programs are analyzed for procedure calls, types, and parameters, and how recursion is detected and handled.
5. **Automatic Instrumentation:** Details the instrumentation, outlines the algorithm used, and gives examples of code generated.
6. **Automatic Parallelization:** Describes the details of the parallelization, presents the algorithm used, and gives examples of code generated by the system and performance measurements of parallelization strategies.
7. **Automatic Selection of Strategies :** Explains how parallelization strategies are automatically selected after a sequential test run and describes the main script's algorithm.
8. **REAPAR Results :** Briefly summarizes the benchmarks' speedups and results of strategy selection and compares the performance to related systems.

Throughout the report, scientific results such as speedups obtained and the effects of parallelization strategies are only mentioned where appropriate to avoid redundancy. For an in-depth discussion, please refer to the PhD thesis [Hän98] (only available in German, sorry) and the forthcoming paper.

For a pure user, reading chapter 2 is sufficient, maybe supplemented by chapter 3 for trouble shooting hints. The remaining chapters focus on the internal algorithms and details that allow an advanced user to gain further insights and obtain more performance from the system.

Chapter 2

Using the System

This chapter describes how the system is used to instrument recursive programs with profile generating code, how thread-parallel programs are automatically derived by source code transformation, and how parallelization strategies are automatically chosen.

2.1 Quick Start

For the impatient, here is what to do in a nutshell — the system just requires a Sun multiprocessor machine running Solaris 2.5.1, the Solaris thread package and a C compiler.

To install the system:

- Unpack the archive reapar.tar.gz:
  `gunzip -c reapar.tar.gz | tar xvf -`
- Compile and install the system components:
  `make`
- Make sure Perl 5.x is installed in `/usr/local/bin/perl` or edit the `*_program` and `reapar` scripts in the `bin` directory accordingly.
- Make sure the `bin` subdirectory is in your search path or copy the files contained in it to a place that is , e.g., your `$HOME/bin` directory.
- Place `profile.c` and `profile.h` in the same directory as the programs to be instrumented resp. parallelized. Symbolic links are OK, too.

To obtain the recursion profile from a program:

- Instrument the program:
  `instrument_program -out` *instrumented.c myprogram.c*
- Compile and link the result:
  `gcc -g -o` *instrumented instrumented.c* `profile.c`
- Run the program:
  instrumented
- The program will print the recursion profile to standard output after any other output it generates.

To parallelize a program by introducing threads at recursive calls:

- Parallelize the program:
  `parallelize_program -out` *parallel.c myprogram.c*

- Compile and link the result:
  `gcc -g -o` *parallel parallel.c* `profile.c -lthread -D_REENTRANT`
- Run the program:
  parallel
- The program will run using the maximum number of processors available. It will output REAPAR specific information gathered during runtime in `"BEGIN REAPAR ..."` and `"END REAPAR ...` blocks which can easily be filtered out automatically.

To perform instrumentation, parallelization, and automatically choose an appropriate Parallelization Strategy:

- Submit your source code, the number of processors you want the resulting program to run on (e.g. 4), and the sample problem to the REAPAR system:
  `reapar` *myprogram.c* 4 "*sample parameters*"
- The system will parallelize your program automatically and generate a parallelized executable — see section 2.8 for details.
- Run the resulting parallel program with your input set:
  myprogram_parallel "*input parameters*"

Please check `http://wwwipd.ira.uka.de/~haensgen/reapar` for updates.

2.2 Environment

The methods described in this report are implemented as Perl 5 [WC97] programs that perform the required transformations. Helper scripts are written in `csh` and performance critical components such as the strategy selection are written in `C`. Programs to be parallelized must be submitted as ANSI C [KR88] source code which is then automatically instrumented and enhanced with thread calls to exploit parallelism in recursions. The system does not include a conventional compiler, it just operates on the source code.

The target architecture are Sun shared memory MIMD machines such as Enterprise Servers or multi-processor SPARCstations running Solaris. The programs generated by the system can be directly linked with the Solaris thread library [Sun94] to produce code that runs in parallel.

2.3 Requirements

A program to be handled by the system has to meet the requirements listed below.

Chapter 3 gives background information regarding these restrictions and shows how to recognize, handle, and overcome them in your programs.

Program — the source program's requirements are:

- It has to be written in ANSI C, especially regarding the function headers.
- Recursive procedures may not have data dependencies among recursive calls if they are to be parallelized.
- Recursive procedures to be parallelized have to have the result type `void`. Any results have to be passed through reference parameters.
- The system expects syntactically correct source code, i.e., programs that a C compiler processes without complaints.

Format — the source code has to meet these requirements:

- All `if, else, while, for` constructs need braces enclosing their respective blocks, i.e., `if (a) f(x) else f(y)` is not allowed while `if (a) {f(x)} else {f(y)}` is.
- All relevant program code (all recursive procedures and any code calling them) has to be contained in one file.
- The procedure's header including the parameter list's opening parenthesis must be on one line. The parameters themselves may span several lines.
- Strings must not span multiple lines.

Details

- C Preprocessor directives are ignored, e.g., procedures textually introduced by a `#define` are not recognized.

2.4 System Components

For an idea of how the system's components work together, please take a look at the REAPAR main script's algorithm in figure 7.2 on page 55.

The `tar` file of the system distribution contains the following directories and files (executable scripts marked with a "*", directories with a "/"):

Main directory:	
`Makefile`	Builds all C programs in the `src/` directory and installs them in `bin/`. Also builds the executables in `utilities/`.
`benchmarks/`	Directory containing the nine Benchmarks used to construct and verify the REAPAR system, see below.
`bin/`	Directory with the `perl` and `csh` scripts. Executables of the REAPAR system are installed here, too.
`profile.c`	Recursion profile generator code to be linked with instrumented programs.
`profile.h`	Definitions for profile generation.
`src/`	Directory in which the C sources of REAPAR components reside.
`utilities/`	Directory with several useful scripts and programs, e.g. for creating a graphical representation of a recursion tree or measuring the performance of all parallelization strategies.

`utilities/` Directory:	
`Makefile`	Builds the programs (called by main `Makefile`).
`do_plots`	Generates Encapsulated PostScript plots from measurement series using `gnuplot`.
`komplette_messreihe`	Performs a complete set of measurements for all parallelization strategies.
`make_plot_from_messreihe.awk`	Called by `do_plots`.
`tree_graph_output.c`	Generates graphical representation of recursion tree.
`tree_parser.c`	Performs statistical analysis of recursion tree.

`src/` Directory:

`Makefile`	Builds the programs (called by main `Makefile`).
`evaluate_profile.c`	Performs the depth strategy selection heuristics on the profile.
`simulation.c`	Performs thread simulation for given strategy, CPU number and recursion tree.

`bin/` directory:

`choose_strategy_by_heuristics*`	Chooses a parallelization strategy for coarse grained programs by thread simulation.
`choose_strategy_by_simulation*`	Chooses a strategy for fine grained programs by profile heuristics.
`hierarchies_check_simulation*`	Given a recursion tree, checks if it's profitable to deactivate lower recursion hierarchies.
`instrument_program`	Performs program instrumentation with profile generating code and (optionally) complete recursion tree recording.
`parallelize_program`	Inserts thread generating code for recursive procedures, resulting in parallel execution.
`reapar`	Main script — automatically instruments a program, examines its execution profile, chooses a parallelization strategy and parallelizes it.
`set_strategy_parameters*`	Activates the given parallelization strategy in a program's source code.

2.5 Benchmarks

The following table gives a brief overview over the benchmarks supplied with the REAPAR system in the `benchmarks/` directory. For a closer discussion, please refer to the PhD thesis [Hän98].

`Makefile`	Runs `reapar` on all the benchmarks with sample problems in the <1 minute range.
`barnes.c`	Galaxy simulation, based on the original source code by J. Barnes [BH86]
`bitonic.c`	Bitonic sorting, based on Olden code [Car96]
`eigenvalue.c`	Computation of Eigenvalues of symmetrical tridiagonal matrices, based on code by S. Chakrabarti et al. [CRY94]
`fractal.c`	Fractal computation by recursive heuristics (own code).
`heat.c`	Heat diffusion simulation, based on Cilk code [Sup97]
`knapsack.c`	Solution to the 0/1 knapsack problem, based on Cilk code
`magic.c`	Computes number of solutions of magic squares, based on Cilk code
`power.c`	Power pricing simulation in tree network, based on Olden code and S. Lumetta's Split-C work [LML+95]
`queens.c`	Computes all solution to the n Queens problem, based on Cilk code

2.6 Instrumentation

The following two sections, Instrumentation and Parallelization, are important only if you want to perform these operations manually. By default, the main script `reapar` described in section 2.8 handles all these steps automatically, but it is a prototype with some limitations, so a knowledge of what is going on under the hood can be helpful.

The program is instrumented by calling `instrument_program` and supplying the name of the source code file to be instrumented. If the environment variable `IP_OUT` is set or the `-out` option is given, the resulting program is written to that file, else it appears on standard output. The following parameters are recognized — unknown options cause the Instrumentation to print the available options and exit:

```
usage: instrument_program [options] filename

       Instruments ANSI C source to collect recursion information profiles.

       Options:
       -detail N    Output detail level. 0=none, 1=informative (default),
                    2=verbose, 3=very detailed, 4=chatterbox
       -degree N    Set max. profile branching degree to N (default = 12)
       -depth N     Set max. profile recursion depth to N (default = 90)
       -noisy       Same as -detail 3
       -autonostats Automatically annotate NOPARALLEL procedures NOSTATS
       -noprofile   Only generate additional parameters but no profile
       -out F       Write instrumented program to file F (default = stdout)
       -quiet       Same as -detail 0
       -rchunk N    Sets chunk size for tree recording (default = 100000)
       -record      Record the complete recursion tree for later analysis
       -time        Get timings for each recursion subtree (implies -record)
       -verbose     Same as -detail 2
```

Branching degree and depth describe the maximum possible number of recursive branches in a single invocation of a recursive procedure and the maximum recursion depth. Default values are 12 and 90, respectively.

By specifying the `-noprofile` option, you can disable the profile information generation, leaving just the addition of depth and branching degree parameters. This is useful mainly in combination with parallelization as described later.

Using the `-record` option adds code that records the complete recursion tree of all recursive procedures over all iterations. It can only be used for sequential runs and adds around 9% to the runtime of the program, more for very fine grained programs. `-time` additionally measures the CPU time each subtree takes to execute in milliseconds. The `-rchunk` option lets you increase the allocation chunk size for the tree recording in case the default value is not sufficient.

The `-autonostats` option automatically inserts a `NOSTATS` annotation (see below) for each procedure that isn't parallelized, thus collecting profile information only for parallel procedures.

The resulting output file has to be linked with the profiling code, e.g.,

```
instrument_program -out instrumented.c program.c
gcc -g -o instrumented instrumented.c profile.c
```

Remember to add any include/header files and libraries your program requires to the above commands.

Running the instrumented program now results in a runtime profile output when the program exits. In the profile, the number of times every branching degree was used at each recursion depth is given for each procedure. Additional information about user and system time consumed as well as the procedures for which information was collected is given, too. All information blocks are in "`BEGIN REAPAR ... END REAPAR ...`" brackets and can thus be filtered out automagically, either to extract REAPAR information or to remove it in order to obtain the original program output. A sample profile output might look like this:

```
BEGIN REAPAR RSCINFO
Wallclock time = 19.76 seconds
User time      = 19.76 seconds
System time    = 0.06 seconds
END REAPAR RSCINFO
BEGIN REAPAR PROCINFO
Procedure ( 1) LoopBody_h           at line  205 calls :LoopBody_h:
END REAPAR PROCINFO
BEGIN REAPAR PROFILES
Profile:
  233 - LoopBody_h
        (  0:    0    0    1    0  sum =    1)
        (  1:    0    0    2    0  sum =    2)
        (  2:    0    0    4    0  sum =    4)
        (  3:    0    0    8    0  sum =    8)
        (  4:    0    0   16    0  sum =   16)
        (  5:    0    0   32    0  sum =   32)
        (  6:    0    0   64    0  sum =   64)
        (  7:    0    0  128    0  sum =  128)
        (  8:    0    0  256    0  sum =  256)
        (  9:  124    0  388    0  sum =  512)
        ( 10:  776    0    0    0  sum =  776)
        ( 11:    0    0    0    0  sum =    0)
        (sum:  900    0  899    0  SUM = 1799)
[...]
END REAPAR PROFILES
```

The profile entry at row r column c shows the number of times a recursion of degree c (i.e., c branches) occured at recursion depth r. Both c and r start at 0. In our example, the possible degrees range from 0 to 3 and depths from 0 to 11. For example, the "124" in line "9" means that at recursion depth 9, the branching degree 0 (i.e., a leaf with no further branches at all) occurred 124 times. Vertical and horizontal sums allow for an easier overview.

Thus, the above profile reads as follows: The recursive procedure LoopBody_h at line 205 has been invoked once from the outside, since the number of entries at recursion depth 0 is one. It just branches twice or never (degree 2 or 0) and reaches recursion depth 10. The number of leafs, i.e., branching degree 0, is 900. Leafs appear just at the last two recursion levels, and the number of nodes (branching degree 2) at recursion depth d is 2^d, indicating a perfect binary tree for recursion.

The use of the `-record` option additionally outputs the recursion trees of all recursive procedures in brief ASCII notation. Output might look like this:

```
BEGIN REAPAR TREES
Procedure number 1 = LoopBody_h

Recursion Tree for 'LoopBody_h', Number 1, iteration 0:

(1(1(1(1(1(1(1(1(1(1(1)(1))(1(1)(1)))(1(1(1)(1))(1)))(1(1(1(1)(1))...
...
END REAPAR TREES
```

Here, the recursion tree for the procedure `LoopBody_h` is shown — "(" represents a step down the recursion tree and ")" exits the current recursion. The procedure's internal number is 1 and it only calls itself recursively since there are no other numbers in the tree. There is just one iteration (call of the procedure from the non-recursive outside).

Details and tools to analyze this output are described in more depth in section 3.1.6. The use of subtree execution time measurement with the `-time` option is explained in section 3.1.7.

2.7 Parallelization

`parallelize_program` is used to derive a thread-parallel program from a recursive C source. If the environment variable `PP_OUT` is set or the `-out` option is given, the resulting program is written to that file, else it appears on standard output. The following parameters are recognized — unknown options cause the Parallelization to print the available options and exit:

```
usage: parallelize_program [options] filename

       Parallelizes recursive ANSI C source with thread calls.

       Options:
       -detail N    Output detail level. 0=none, 1=informative (default),
                    2=verbose, 3=very detailed, 4=chatterbox
       -degree N    Set max. profile branching degree to N (default = 12)
       -depth N     Set max. profile recursion depth to N (default = 90)
       -noisy       Same as -detail 3
       -autonostats Automatically annotate NOPARALLEL procedures NOSTATS
       -noprofile   Only generate additional parameters but no profile
       -out F       Write parallel program to file F (default = stdout)
       -quiet       Same as -detail 0
       -verbose     Same as -detail 2
```

By specifying the `-noprofile` option, you can disable the profile information generation, leaving just the addition of depth and branching degree parameters — this option is directly passed to the instrumentation phase. Use this option if the profile itself is of no concern and performance is critical (profile collection imposes a 0.1% to 4.6% runtime penalty depending on the program's granularity).

The `-autonostats` option automatically inserts a `NOSTATS` annotation (see below) for each procedure that isn't parallelized, thus collecting profile information only for parallel procedures.

In addition to these options, the following annotations can be inserted into the target program's source code to influence the system's decisions:

`/* NOPARALLEL */`
: Annotates that the recursive procedure in which the comment appears should not be parallelized at all, e.g., because its recursive branches are not independent from each other, or because the work done in each branch is too fine-grained to justify parallelization. By default, all recursive procedures with return type `void` are parallelized.

`/* NEEDRESULTS */`
: By default, all threads generated in a procedure are joined just before returning from the procedure. If data computed by the recursive calls is needed earlier in the procedure, this annotation has to be inserted to mark the place where results are needed, forcing a wait for all the procedure's child threads.

`/* NOTHREAD */`
: This annotation prevents the following procedure call from being parallelized with threads. Use it if there is a static number of recursive calls in a procedure and you want to prohibit unnecessary thread creation for the last such call.

`/* NOSTATS */`
: If a recursive procedure is not interesting or would produce an enormous recursion tree, you can disable profile and recursion tree generation for it using this annotation. When recording trees, a Nostats-annotated procedure must not call another recursive procedure that still records statistics — the system recognizes this condition and exits with an error message if it is not met.

These annotations are explained in more depth in sections 3.2.2 through 3.2.5.

The resulting output file is automatically instrumented after parallelization to supply the necessary depth/branch variables. It then has to be linked with the profiling code, e.g.,

```
parallelize_program -out parallel.c program.c
gcc -g -o parallel parallel.c profile.c -lthread -D_REENTRANT
```

Remember to add any include/header files and libraries your program requires to the above commands.

When the parallelized program is run, it generates threads for recursive calls according to some parallelization strategy (i.e., a decision when to perform a recursive call sequentially and when to use a thread). The strategies are explained in chapter 6.4.

By default, the program uses the maximum number of processors available on the machine. This can be adjusted by setting the environment variable `T_CPUNUM` to some other value between 1 and the processor number.[1]

The following sample output shows how the processor number, the selected strategy, the parallelized procedure(s) and the number of threads generated are printed — in addition to any other normal program output not shown here:

[1] This directly affects the Solaris concurrency level, i.e., the number of LWPs used.

```
> parallel_eigenvalue u 900

BEGIN REAPAR PARINFO
4 processors online
strategy general   = ACTIVE_N (N=3)
Parallel procedure ( 1) LoopBody_h          at line  205
END REAPAR PARINFO
BEGIN REAPAR THREADINFO
total number of threads created = 40
END REAPAR THREADINFO
[...]
```

2.8 Automatic Strategy Selection and Parallelization

One of the goals of the REAPAR project was to deliver a turnkey system in which the user just submits C source code and sample input data to the system and gets a parallelized program in return.

The `reapar` script does just that. Given a C program, it instruments the source, compiles it, runs it on the given input data, analyzes the execution profile, chooses a parallelization strategy and parallelizes the program accordingly. `reapar` takes the following parameters and environment variables:

```
usage: reapar [-detail <n>] [-keepfiles] <source.c> <cpus> <parameters>

       Instruments, executes and parallelizes the given ANSI C program.

        source.c        File containing all the program's C source
        cpus            Number of CPUs to use for parallelization
        parameters      Any parameters the program takes for the sample run
                         (rest of the command line)
        n               Level of detail (0=none, 1=informative (default),
                         2=verbose, 3=very detailed)

       If the -keepfiles switch is used, temporary files are not
       deleted after use, e.g. for examining the profile manually later.

       The contents of the environment variable R_COPTIONS are used as
       additional arguments to the compiler call for generating the
       executable (e.g. "othercode.c -lm").
       Likewise, the contents of R_SOPTIONS are passed to the simulation,
       e.g. for telling it a Nothread annotation is active ("-b A1" etc.)
```

Internally, this script calls both Instrumentation and Parallelization as well as the C compiler and the Strategy Selection scripts mentioned before. It can be configured to match your setup by editing the first few lines where e.g. the compiler call and the granularity factor are defined.

Parallelizing a program can be as easy as entering

```
reapar benchmarks/eigenvalue.c 4 "u 1500"
```

This parallelizes the Eigenvalue benchmark for the problem "1500 uniformly distributed eigenvalues", resulting in the following informational output:

```
=== REAPAR: Instrumenting 'benchmarks/eigenvalue.c'===
Writing output to file 'benchmarks/eigenvalue_instrumented.c' ...
=== REAPAR: Compiling 'benchmarks/eigenvalue_instrumented.c' ===
=== REAPAR: Executing 'benchmarks/eigenvalue_instrumented u 1500' ===
   Overall execution time 53.91 seconds,
   2999 leafs in first recursive procedure
-> Coarse grained program, more than 1.00 ms/leaf (17.98)
=== REAPAR: Performing coarse grained strategy selection ===
=== REAPAR: Instrumenting 'benchmarks/eigenvalue.c' with tree recording===
Writing output to file 'benchmarks/eigenvalue_recorded.c' ...
=== REAPAR: Compiling 'benchmarks/eigenvalue_recorded.c' ===
=== REAPAR: Executing 'benchmarks/eigenvalue_recorded u 1500' ===
Performing simulations for 4 CPUs with recording
  /tmp/reapar_profile_20742 and options ''
[...]
Overall ranking:
Rank 1: Depth 3   (1503 steps)
Rank 2: Active 1  (1751 steps)
Rank 3: Keep 2    (2589 steps)
=== REAPAR: Parallelizing 'benchmarks/eigenvalue.c' ===
Writing output to file 'benchmarks/eigenvalue_parallelized.c' ...
=== REAPAR: Setting parallelization strategy 'DEPTH 3' ===
=== REAPAR: Compiling 'benchmarks/eigenvalue_parallelized.c' ===
=== REAPAR: Your parallel program is ready for execution! ===
-rwxr-xr-x   1 haensgen      37728 Mar 06 15:45 benchmarks/eigenvalue_parallel
```

Since the `reapar` script is just a proof of concept, there are some limitations which are explained in more depth in section 3.3. For more information on how the script works and which other parts of the REAPAR system are involved, please refer to section 7.4.

Chapter 3

Advanced Features

This chapter explores the system's features in more depth and gives advice on how to recognize and handle its limitations.

3.1 Options Explained

The following sections describe the use of the instrumentation's and parallelization's options in more detail and give examples of how and when to apply them. The main `reapar` script is described in a separate section later.

3.1.1 `-detail` N

Sets the output detail level. The effects of the different levels are:

`-detail 0` or `quiet`: Does not produce any output except for error messages and final program output confirmation.

`-detail 1` (default): Also gives information on the system's progress and displays the procedures, calls and recursions detected.

`-detail 2` or `verbose`: Additionally shows the current output detail level and parameter settings, prints the source code lines as procedure and call identification progresses as well as procedures, declarations, variables, calls and annotations found, and thread resp. profile addition phases.

`-detail 3` or `noisy`: Additionally outputs pattern matches when detecting procedures and calls, more detailed progress information, internal procedure use (skipping to delimiters etc.), and even more thread/profile addition information.

There is also a detail level 4, which outputs a mass of internal information such as the source code lines being read or all code insertions performed, and is probably not of much use for anyone who doesn't want to dive into the system's source code.

Program parallelization calls the Instrumentation phase internally, but lowers the detail level by one, i.e., calling Parallelization with detail level 2 results in Instrumentation being called at level 1. This avoids verbose output that is not that interesting for parallelization itself.

All detail levels output the transformed program's source code if neither the `-out` option nor the respective environment variables are set (see section 3.1.5).

3.1.2 `-degree` N and `-depth` N

By default, programs are assumed to have a maximum branching degree of 12, i.e., a recursive procedure calls performs no more than 12 recursive calls, and a maximum depth of 90, i.e., there are no more than 90 levels of recursion.

If the number of branches or the recursion depth are known in advance, e.g., the program just branches twice or never, these options can be used to reduce the size of the profile data structure. However, since each profile entry just takes a `long` plus some bytes of information for each depth level, the profile's memory requirements are not very high, anyway.

Increasing the depth or degree parameters is necessary if the branching degree or the depth are known to be higher than the default settings. An indication of this case may be core dumps — because profile generation is very time critical, no boundary checks are performed, leading to problems if subscripts get out of range during program execution.

Recursion tree recording as described in section 3.1.6 is done completely dynamically since it is inherently slower anyway, thus it is not affected by these parameters.

3.1.3 `-noprofile`

Using this option disables profile information gathering, i.e., no array of branch counts and depths is generated at all. However, Instrumentation and Parallelization still add a depth parameter to each recursive procedure and introduce a branch counter. These variables are necessary for parallelization to work (keeping track of threads generated, joining them, recognizing the recursion depth for parallelization strategies that use it, etc.).

You may gain some speed by disabling profile generation, depending on the program's granularity. Additionally, accessing the profile information in memory may result in suboptimal cache usage. If you have no use for the profile and want to obtain maximum speed, this option can save you 0,1% to 4,6% of the sequential runtime. Furthermore, compiling and linking with the profile code (`profile.c` and `profile.h`) is no longer necessary.

3.1.4 `-autonostats`

If this option is used, a `NOSTATS` annotation is automatically inserted before any procedure that isn't parallelized, thus collecting profile information only for parallel procedures. This can save much space and time — e.g., the Barnes Hut benchmark with recursion tree recording runs 2.6 times slower than the uninstrumented version and produces 160MB of output if all recursive procedures are instrumented. However, with the autonostats option and parallelization of just the upper recursion hierarchy, output shrinks to manageable 5MB and execution time only increases by 4%!

3.1.5 `-out` F

This option just sets the name of the output file for the program generated. Parallelization additionally uses a temporary file called `tmp_`*program_name*_*PID* in the current directory on which it then performs the Instrumentation. This file is deleted after successful parallelization.

Using this option overrides the environment variables `IP_OUT` resp. `PP_OUT`.

3.1.6 `-record` and `-rchunk` N

Often, even the programmer has no intuition how the tree of a procedure's recursive calls will look like, or the intuition is wrong. To gain better insights in a program's behavior, program Instrumentation offers this option to record all recursion trees at runtime. Even more important for REAPAR, tree recordings are the basis of thread simulation as described in section 7.3. Recording only works on sequential machines since the tree construction would result in write conflicts when performed concurrently. All recorded recursion trees are output at the end of the program for all iterations over the recursive procedures, e.g., for a galaxy simulation over 20 time steps that is internally recursive, 20 recursion trees are printed.

The resulting output can be converted to several graphic formats using the `tree_graph_output` program which takes the following options:

```
tree_graph_output - reads profile-generated tree from stdin,
   writes tree graphics as commands to stdout.
   Usage: tree_graph_output [options] < treefile > comfile
   Options:
     -comdraw  Output in comdraw format  (also '-c')
     -pstricks Output in pstricks format (also '-p', default)
     -x n      Sets x spacing of nodes to n (default=20)
     -y n      Sets y spacing of nodes to n (default=12)
     -s n      Sets node diameter to n      (default= 2)
     treefile  ASCII representation of recursion tree, e.g.
               "(()())" for simple binary tree, may be annotated
               with procedure numbers and subtree timings
```

Comdraw output can be fed to the standard input of the `comdraw` tool freely available from `http://www.vectaport.com/ivtools/`, and PsTricks output can directly be used in the TEXpackage of that name. The outputs shown below have been rotated 90^o to place the tree's root at the top instead of its original left hand side placement. For large trees which only have one type of node, the node size can also be set to zero for a less cluttered output.

To give an impression what kind of results can be obtained, here are three examples:

1. **Single recursive procedure:** If every recursive procedure just calls itself recursively, there will be a recursion tree for each iteration over each procedure, representing exactly this procedure's recursions. An example for this structure where `foo()` calls `foo()` calls `foo()`... is:

```
void foo(int i) {
  if (i>100) {
    foo(i/2);
    foo(i/3);
    return;
  }
  if (i>10) {
    foo(i-30);
  }
}
```

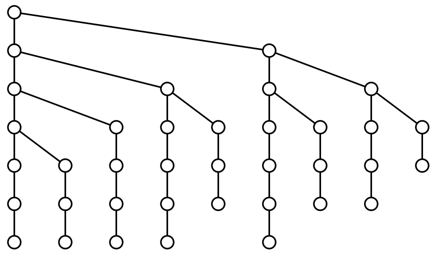
For this program, the call `foo(1000)` results in the instrumentation output

```
BEGIN REAPAR TREES
Procedure number 1 = foo

Recursion Tree for 'foo', Number 1, iteration 0:

(1(1(1(1(1(1(1)))(1(1(1))))(1(1(1(1)))))(1(1(1(1(1)))(1(1(1)))))(1(1(1(1(1))))
(1(1(1))))(1(1(1(1)))(1(1)))))
END REAPAR TREES
```

There are no other calls to `foo()` from the outside, so this is the only iteration. Analyzing this tree produces the following graphical tree representation, which makes the program's recursion structure quite clear:



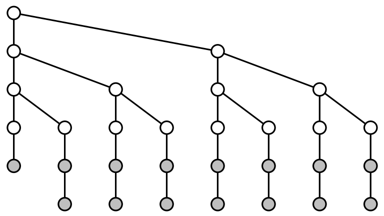
2. **Hierarchically recursive procedures:** Often, there are several levels of recursion, i.e., a procedure `a()` calls itself until some condition occurs and then passes the subproblem to some procedure `b()` that works on it recursively, too. There are no calls back to `a()`. An example for this structure is the Barnes Hut galaxy simulation, where a top-level recursion traverses the tree of bodies and for each single body there is a recursive call that computes the forces on that body.

   The following code shows a similar structure:

```
a(int x1, int x2) {
   if (x2-x1<2) {
      b(x1);
      return;
   }
   a(x1, (x1+x2)/2);
   a((x1+x2+1)/2, x2);
}
```

```
b(int x) {
   if (x>1)
     b(x-30)
}
```

   Program Instrumentation handles calls of recursive procedures by other recursive procedures in the intuitive way — the first call from the non-recursive outside starts the recursion tree. After that, each recursive call contributes to the common recursion tree, no matter in which procedure involved in the recursion it occurs. The same semantics hold for more than two levels of recursion, too. Thus, the resulting recursion tree for the call `a(1,10)` in our example looks like this:



   In this case, two shades of gray are automatically used in the graphical output to tell the two different procedures apart — procedure `a()` has white nodes, `b()` has gray ones.

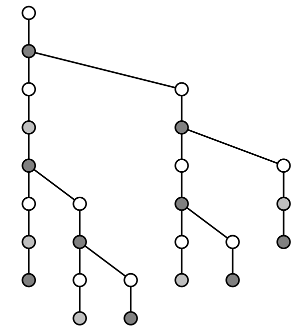
3. **Mutually recursive procedures:** This is the most challenging case to visualize without Instrumentation's help — a procedure a() calling b() or c() calling a() again, possibly over more intermediate procedures:

```
a(int i) {
   if (i%2) {
     b(i/2);
   } else {
     c(i/3);
  }
}
```

```
b(int i) {
  if (i>1) {
    c(i-10);
  }
}
```

```
c(int i) {
 if (i>1) {
    a(i/3);
    a(i/3-1);
  }
}
```

   As in the case of hierarchical levels of recursion, the recursion tree of the first procedure called from the outside of this mutual recursion is passed to all other procedures called, resulting in a recursion tree that covers the mutual recursion as a whole. The graphical tree representation resulting from the analysis of a(1000) is:



   Again, different shades of gray stand for the different procedures.

The recursion tree is built up in memory dynamically during runtime. Initially, room for 100 000 nodes/leafs is allocated by default, and after each iteration over a recursive procedure, further nodes/leafs are allocated if necessary. Inside an iteration, no boundary checking is performed for reasons of speed. Therefore, in programs that generate trees with more than 100 000 nodes in a single iteration, you have to increase this number by using the instrumentation's -rchunk *N* option. Otherwise, you can expect segmentation violation errors.

As mentioned above, the tree recording takes an additional 1% to 160% runtime for the REAPAR benchmarks, the finer grained the program the more. Fortunately, fine grained programs only benefit from the Depth parallelization strategy anyway which can be selected automatically just by examining the profile, not the recursion tree.

3.1.7 -time

The exact recursion tree obtained by -record already reveals a wealth of information. However, for manual fine tuning it can be important to know how much time exactly is spent in each subtree, e.g., if the complexity of recursive procedures varies greatly with their input or recursion depth. For such procedures, the recursion tree alone does not give enough information on how well a later parallel execution could be balanced among processors — if one single procedure invocation requires as much computation as a hundred invocations of another recursive procedure, timings must be taken into account.

The REAPAR system's instrumentation component offers the option `-time` for such cases. Using this option implies `-record`. When timing is activated, the CPU time used by each procedure invocation is measured using the `clock()` system call. CPU time at entering the procedure is subtracted from the time at leaving it, giving a measure of the time spent inside its subtree. The instrumentation output reflects this, as the following example shows:

```
(1:740(3:310(1:100(2:60(3:40(1:10(2:10(3:0)))(1:10(3:10(1:0(2:0))(1:0(3:0))))))
(1:70(3:30(1:10(3:0(1:0(2:0))(1:0(3:0))))(1:0(2:0(3:0)))))))
```

CPU time is given milliseconds, appended with a ":" to the procedure numbers in the output. Subsequent analysis using the `tree_parser` program yields:

```
Tree indented = (proc: time nodes/leafs)
 A:740ms 19/6 C:310ms 18/6 A:100ms 9/3  B:60ms 8/3 C:40ms 7/3 A:10ms 2/1 B:10ms 1/1 C: 0ms 0/1
                                                              A:10ms 4/2 C:10ms 3/2 A: 0ms 1/1 B: 0ms 0/1
                                                                                    A: 0ms 1/1 C: 0ms 0/1
                           A: 70ms 8/3  C:30ms 7/3 A:10ms 4/2 C: 0ms 3/2 A: 0ms 1/1 B: 0ms 0/1
                                                                         A: 0ms 1/1 C: 0ms 0/1
                                                   A: 0ms 2/1 B: 0ms 1/1 C: 0ms 0/1


Tree graphically (procs as chars)=
   A--C--A--B--C--A--B--C
         |        A--C--A--B
         |              A--C
         A--C--A--C--A--B
               |     A--C
               A--B--C
```

Under Solaris, the effective timer resolution is 10ms, meaning that small computations cannot be measured exactly. Also, the CPU timer overflows at around 30 minutes, so only computations that take less time can be measured at all.

Since no direct benefits or better strategy selection can be derived from this information, timings are not used by the REAPAR system itself. Their purpose is to provide the interested user with additional information.

The program `tree_parser` supports the following options:

```
tree_parser - parse recorded recursion tree from stdin,
optionally with recorded procedure numbers and timings

Program options:
  -d  output detailed information
  -g  output graphical tree
  -gv ditto verbose (procedure name tags)
  -s  output statistics
  -sv ditto verbose (sizes of subtrees)
  -t  output tree
  -tv ditto verbose (timings etc)
```

It performs a detailed statistics analysis of the given recursion tree, including the averages and deviations of (sub)trees and leafs, and optionally outputs the tree as ASCII graphics as shown above. Additionally, it recommends depths for the Depth parallelization strategy, but the Simulation described in chapter 7.3 is a more recent development and as such better suited for this job.

3.2 Understanding and Overcoming Restrictions

This section explains the reasons for the system's limitation mentioned in section 2.3 and shows ways of avoiding them.

3.2.1 Correct Source Code

The system handles only syntactically correct input source code, i.e., code that a C compiler translates without complaints. There is no way around this requirement — the system cannot be expected to produce any sensible results from incorrect source code.

3.2.2 `NOPARALLEL` Annotation and Hierarchy selection

By default, all recursive procedures in the program are parallelized. Sometimes, this is not desired, e.g., if

- a procedure contains data dependencies (see also 3.2.8) such as writing to a global variable, or a procedure constructs a recursive data structure and relies on some order of execution, or
- a procedure is potentially parallel but the work performed is so fine-grained that parallelization and thread generation overhead outweigh the benefits.

In any of these cases, just insert the annotation `/* NOPARALLEL */` inside the procedure and the system will not parallelize it; adding the annotation just before the procedure works, too.

The fine grained case can be identified by taking a look at the program's profile and estimating the execution time of one recursion leaf — if the total program's sequential runtime is five seconds and the recursion tree has 50 million leafs, parallel slowdown instead of speedup is very likely.

To help the user select procedures that can be annotated Noparallel, the shell script `hierarchies_check_simulation` simulates a given recursion tree using the Always parallelization strategy. If the script detects enough parallelism in the upper layers of a recursion hierarchy, it recommends using the Noparallel annotation on the lower recursion layers, as the following example shows:

```
Analyzing recursion tree 'recorded_barnes_16384' on 16 CPUs
for potential non-parallelization of recursion hierarchies

Recursion tree starts with procedure computesubtree (B)
computesubtree calls walksub (C)

Simulating with all 2 procedures parallel (BC)
-> Simulation predicts enough parallelism if all procedures are parallelized
   (largest sequential chunk: 0.49% of all nodes and 0.20% of all leafs)

Simulating with first procedure parallel (B)
-> Simulation predicts enough parallelism if the procedure
   'walksub' is executed sequentially
   (largest sequential chunk: 3.92% of all nodes and 5.13% of all leafs)

   -> Recommend /* NOPARALLEL */ annotation for 'walksub'
```

3.2.3 NEEDRESULTS Annotation

Some recursive programs perform calculations that are just used locally inside each procedure's invocation and explicitly passed back to the caller later, or that affect independent parts of some global data structure. For these, the system's default behavior (joining any threads generated within a procedure call just before returning from the procedure) is perfect.

However, other programs use some recursive calculations and then use the results of those before returning. These programs can be parallelized successfully by performing the recursive calls concurrently, but for the program to run correctly, the user has to make sure that all threads are joined and thus their results are available before that data is used.

To do so, insert the `/* NEEDRESULTS */` annotation at the source code line just before the results of previous recursive calls are needed — you can think of Needresults as a statement that forces all data from former recursive calls to be present. Using this annotation as late as possible in the code improves performance since it allows the recursive computations to overlap with the remaining code. See section 3.2.7 for further examples.

You can easily notice when a program needs this annotation: The results of the parallel program are incorrect or the program even crashes, since data with wrong values or even uninitialized data is used which would have been set to a valid value in a sequential run, but which is not valid in parallel since the recursive call writing the data has not finished yet and is still running in parallel.

3.2.4 NOTHREAD Annotation

By default, each recursive call is parallelized using threads, which is appropriate for programs with a dynamic branching degree. However, for procedures that branch statically and do not perform much computation between the recursive calls and the place where their results are needed, it leaves room for optimization: The last recursive call can be performed sequentially, i.e., instead of generating a new thread, the last call can be executed by the thread that originally invoked the procedure itself.

To prevent unnecessary thread creation for a recursive call in such cases, insert the `/* NOTHREAD */` annotation before the call or in the same line as the call. The following example illustrates this:

```
foo(int a) {
   ...
   foo(x);
   foo(y);
   foo(z);   /* NOTHREAD */
   ...
}
```

Here, the procedure `foo()` branches statically three times, i.e., the branches occur unconditionally and without any loop. Therefore, the thread that executes the current invocation of `foo()` can perform the `foo(z)` call, too, instead of spawning a new thread for it and then sitting around waiting for the spawned threads to complete.

For irregular programs using the Active N or Keep N parallelization strategy (see section 6.4), this can lead to performance improvements of 5%-20% and make the choice of a good parallelization strategy easier.

To help the user in selecting procedures that might benefit from this annotation, the program `evaluate_profile.c` (besides its main function, namely to choose a Depth parallelization strategy) makes the corresponding recommendation if a procedure always branches n times or never, as the following output shows (marked "«<"):

```
226 - compute_area
      (  0:     0    0    0    0     1    0    0  sum =     1)
      (  1:     0    0    0    0     4    0    0  sum =     4)
      (  2:     0    0    0    0    16    0    0  sum =    16)
      (  3:     8    0    0    0    56    0    0  sum =    64)
      (  4:    50    0    0    0   174    0    0  sum =   224)
      (  5:   177    0    0    0   519    0    0  sum =   696)
      (  6:   633    0    0    0  1443    0    0  sum =  2076)
      (  7:  1801    0    0    0  3971    0    0  sum =  5772)
      (  8:  4891    0    0    0 10993    0    0  sum = 15884)
      (  9: 43972    0    0    0     0    0    0  sum = 43972)
      ( 10:     0    0    0    0     0    0    0  sum =     0)
      (sum: 51532    0    0    0 17177    0    0  SUM = 68709)
```

```
> ./evaluate_profile -a < profile_fractal1_1024_1024
profiled procedure 'compute_area' at source line 226
maximum degree = 4, maximum depth = 9
profile covers 1 iterations
recommend using the NOTHREAD annotation for branching degree 4          <<<
 since degree is always 0 or 4.                                          <<<
performing Average Subtree analysis for 5 CPUs

analyzing depth 1:
all in all, the average subtree has 17178.0 nodes.
that's  25.00% of the tree's nodes
-> not recommended for 5 CPUs, average subtree not smaller than 1/5 (20.00%)

analyzing depth 2:
all in all, the average subtree has 4295.0 nodes.
that's   6.25% of the tree's nodes
-> RECOMMENDED: depth  2 for  5 CPUs, average subtree is only 6.25%
```

3.2.5 `NOSTATS` Annotation

For procedures whose profile does not need to be collected, the user can insert this annotation before or inside the procedure. The effect is that neither profile tables nor recursion trees are recorded for the procedure. Only the recursion depth counter is added and passed to recursive procedures called in the annotated procedure. This is useful if

- there are many recursive procedures that are of no interest to the user and clutter the view,
- the user only wants to see the profile of parallel procedures and chooses to disable all other profiles, or
- the lower hierarchies of a recursion tree produce so many nodes and leafs that the recording becomes unmanageable in size and takes a huge amount of computation time.

As mentioned, procedures annotated Nostats must not call other recursive procedures which still collect statistics if tree recording is activated. Otherwise, the data necessary for the recording could not be supplied by the Nostats procedure. The instrumentation recognizes this condition and aborts with a warning. Normal profiles without trees are not affected, the Nostats procedure just does not appear in the profiles.

3.2.6 ANSI C

As stated in the requirements section, the system needs ANSI C source code. The reason is that in ANSI C, procedure headers and declarations are much easier to process automatically than in K&R C. Apart from procedure headers, no ANSI C features are needed, so you can prepare a K&R C program for parallelization by just ANSI-fying its procedure declarations and headers.

3.2.7 No Return Values

Procedures to be parallelized must not have a return value — the system will ignore non-`void` procedures and give a warning message. The reason for this limitation is that it is difficult to automatically derive where results are needed after the result has been computed in a separate thread. For example, take a look at this code fragment:

```
a = foo(b) + bar(c) + foo(d);
```

(the system would have to introduce temporary variables, start the calls as threads, and join the threads at once to compute `a`) or even this piece of code:

```
for (i=0; i<max; i++)
    if (cond(a[i]))
        z = z + foo(a[i])
```

(the system would have to introduce a temporary array of results where depending on the condition function not all members are defined at all).

Therefore, the following two tasks are upon you:

Eliminating Return Values

For all recursive procedures that are to be parallelized, turn the procedure's return value into a reference parameter, e.g.,

```
double myfunction(double a) {
   ...
   return b;
   ...
}
...
x = myfunction(y)
```

$\Longrightarrow$

```
void myfunction(double *result,
                double a) {
   ...
   *result = b;
   ...
}
...
myfunction(&x, y)
```

Adding Temporary Variables

Supply your own temporary variables for results of recursive calls. Here, you can also use the `/* NEEDRESULTS */` annotation described above to make sure the results are available when you want to refer to them.

```
double foo(double a) {
  ...
  for (i=0; i<max; i++) {
     if (cond(a[i])) {
        z = z + foo(a[i]);
     }
  }
  ...
}
```

⇒

```
void foo(double *result, double a) {
  double r[MAXNUM];
  ...
  for (i=0; i<max; i++) {
     if (cond(a[i])) {
        foo(&(r[i]), a[i])
     } else {
        r[i] = FLAG;
     }
  }
  ...
  /* NEEDRESULTS */
  for (i=0; i<max; i++)
     if (r[i] != FLAG) {
        z = z + r[i];
     }
}
```

3.2.8 Avoiding Dependencies

As stated above, the system can only make use of parallelism in recursive procedures that do not have data dependencies between them: Recursive invocations of a procedure must neither use data generated by parent calls (except when the data used is generated in the parent call before any recursive calls were made), nor values computed by calls at the same level, nor values computed by child calls further down the recursion tree (except when the recursive calls have been joined explicitly before that, see 3.2.3). Also, concurrent use of global data is only allowed when disjunct parts of the data are accessed, e.g., building non-overlapping parts of a picture in parallel recursive calls is OK, but writing to one global variable in each recursion step is forbidden.

If these dependencies occur, you have to use the Noparallel annotation (see 3.2.2) to avoid concurrent writing of data with indeterminable results or usage of uninitialized data. Typical symptoms of data dependencies are wrong or irreproducible results generated by the parallel program or even crashes.

In many cases, the problem can be solved by

- Using local substitutes for global variables — e.g., when a global variable is used as a counter, each recursive call can use its own local counter and return its value to be summed up in the calling procedure.
- Introduce new temporary variables that can be written concurrently and are used later when all results are available (see section 3.2.7).

3.2.9 Applying the C Preprocessor

The REAPAR system does neither recognize nor handle C Preprocessor directives (e.g., `#define`). It just ignores them. Normally, this is no problem and most source code can be used even if it contains directives.

However, it may be necessary to handle directives if

- Code is `#included` that is relevant for the system, e.g., recursive procedure code or forward declarations of procedures that would have to be adapted by the system.

- Conditional compilation (`#if`, `#ifdef` etc.) misleads the system, e.g., procedures that appear to be recursive but are not given the current preprocessor settings, code that has been commented out using `#if`, and so on.

A special form of the second case is incorrect statement skipping — to add code before a statement, the system skips to the beginning of a statement and inserts code there. Since preprocessor directives are not recognized, they are skipped as part of the statement, leading to potential problems. Example:

```
#if something
statement A;     /* "Recognized" as beginning of statement B  */
#endif
statement B;     /* Insert ";" at beginning of line to fix it */
```

When looking for the start of statement B here, the system skips backward until the `";"` at the end of statement A, which leads to errors depending on the outcome of the `#if` directive. As quick fix, just add an empty `";"` before statement B.

The solution to the more general problems mentioned above is to either resolve the preprocessor directives manually (recommended for small changes) or run your source code through the C preprocessor and apply Instrumentation and Parallelization to the resulting source code:

`/usr/lib/cpp -C` *myprogram.c myprogram_cpp.c*
`instrument_program` *myprogram_cpp.c* or
`parallelize_program` *myprogram_cpp.c*

Depending on your configuration, you may have to call another preprocessor or add further definitions required for your program, resulting in command lines such as

```
/export/opt/GNU/lib/gcc-lib/sparc-sun-solaris2.4/2.7.2.1/cpp -D__sparc=1
-C barnes.c barnes_cpp.c
```

The REAPAR system handles the resulting output without difficulties, but it may run more slowly since the preprocessed program explicitly lists all procedures and library calls, dragging down procedure and call recognition. Therefore, for directives that are easily resolved manually (deactivating sections code, including a file, etc.) you may wish to avoid using the preprocessor.

3.3 Automatic Parallelization with `reapar`

The `reapar` script automatically parallelizes your source code for the given sample problem. Since it is a prototype implementation, there are some technical restrictions you should be aware of which are listed in the next sections. All REAPAR sample programs in the `benchmarks` directory already take these into account and run "out of the box".

3.3.1 Files generated and `-keepfiles` option

Given the source code *source.c*, `reapar` generates the following files:

source`_instrumented.`*c*	Source code with instrumentation for profiling added.
source`_instrumented`	Corresponding executable.
source`_recorded.`*c*	Source code with instrumentation for profiling and tree recording added (only generated for coarse grained programs).
source`_recorded`	Corresponding executable.
source`_parallelized.`*c*	Source code with thread parallelization and chosen parallelization strategy added.
source`_parallelized`	Corresponding executable.

Additionally, several temporary files are generated (n stands for the PID of the process executing the `reapar` script):

`/tmp/reapar_tmp_`n	Output of the instrumented program on its sample run.
`/tmp/reapar_profile_`n	Filtered output of the instrumented program, i.e. the profile of the first parallelized procedure for fine grained programs resp. the first recursion tree for coarse grained programs.
`/tmp/reapar_choose_`n	Output of the strategy selection script.

By default, these are deleted after successful parallelization. However, you may wish to keep them for manual analysis as described below. In this case, just use the "`-keepfiles`" option.

3.3.2 Key procedure for Parallelization

To reduce its complexity, the `reapar` script only examines the first parallelized recursive procedure for all its analyses. This means that you have to make sure the procedure to be parallelized is the first recursive procedure in the program. You can achieve this by using the Noparallel annotation on all other recursive procedures or by moving the target procedure's code to the top of the program, adding forward declarations if necessary. `reapar` uses the Instrumentation's `autonostats` option, so procedures not to be parallelized do not generate any profile information at all. If you are not sure about the sequence of procedures in your program, just invoke REAPAR with the "`-detail 2`" option and look at the Instrumentation's output (Instrumentation and Parallelization are called with a verbosity level one lower than `reapar`'s).

Also, the script does not implement profile merging, so only the first procedure in a recursion hierarchy is analyzed. This affects e.g. the Barnes Hut and Bitonic Sort benchmarks. If you want to include the effects of multiple recursive procedures in the parallelization analysis, use the `-keepfiles` option described below, edit the resulting profile removing the "`[...]`" line between the procedures' profile tables and run the `choose_strategy_by_heuristics` script on the resulting profile.

3.3.3 Program Exit Status

`reapar` checks the exit status of all commands it executes to see if any errors occurred. To avoid false reports of failures during execution of the instrumented program, please make sure your program ends with an "`exit(0)`" or "`return(0)`" statement on successful completion. The script reminds you to do so if the instrumented program returns an error code. If you did add the proper statements, there may be a problem with the instrumentation. Please check for possible reasons such as `#ifdefs` as described in the sections above.

3.3.4 Setting proper simulation parameters

If you use annotations such as Nothread in coarse grained programs, you have to tell the simulation about them. You can pass the necessary information to the simulation by setting the environment variable `R_SOPTIONS`. There are two kinds of parameters the simulation takes into account:

`-t` *ABCD...* : Only the procedure $A/B/\ldots$ is simulated as parallel. All remaining recursive procedures that appear in the recursion tree are simulated as sequential as if they were annoted Noparallel.[1]

`-b` *Ax* : Branch number x of procedure A is executed sequentially. This option is used to simulate the effects of a Nothread annotation. Branches are numbered sequentially starting with 0, so e.g. `-b A1` would cause the simulation to behave correctly for the Eigenvalue benchmark which has one parallel recursive procedure whose second branch is annoted as Nothread.

The recursive parallel procedures of a program are represented by upper case letters, `A` standing for the first such procedure, `B` for the second and so on. If there is just one recursive parallel procedure, it is of course `A`. If there are several such procedures, you can identify their corresponding letters by running `reapar` with the `-detail 2` option and looking at the instrumentation's output which shows all recursive procedures detected. Procedure number 1 corresponds to `A`, 2 to `B` and so on.

If you do not set the simulation's options, nothing breaks but the simulations accuracy may be affected. The default behavior is to simulate all branches of all procedures in the recursion tree as parallel.

The Simulation settings for the REAPAR benchmarks with the `-autonostats` option active are as follows (fine grained benchmarks shown in parentheses do not use the Simulation at all but are listed for completeness):

Benchmark	Setting	Comment
Barnes Hut	`-t B`	Only top level recursion parallelized
(Bitonic Sort	`-b A1 -b B1`	Nothread annotations on 2nd branches)
Eigenvalue	`-b A1`	Nothread annotation on 2nd branch
(Fractal	`-b A3`	Nothread annotation on 4th branch
Heat	`-b A1`	Nothread annotation on 2nd branch
(Knapsack	`-b A1`	Nothread annotation on 2nd branch)
(Magic	—	)
Power	`-t A`	Only top level recursion parallelized
(Queens	—	)

[1]Since `reapar` uses the `-autonostats` switch, no statistics are generated by non-parallel procedures anyway. However, if you remove the said switch from `reapar`, you can take sequential parts of the recursion tree into account. In this case, you have to tell the simulation about it as described.

3.3.5 Differences to the PhD thesis results

Since the `reapar` script was written as an add-on after the PhD thesis [Hän98] was finished, there are some small differences between the results of the thesis and what you get when you run the script on the benchmarks directly. The results described in the thesis are based on manually calling instrumentation and parallelization and then analyzing the results using the tools described earlier, while the `reapar` script uses a more direct approach that does not take all possible cases into account. Specifically, `reapar`

- only examines the first parallelizable recursive procedure as mentioned above, i.e., no profile merging of all procedures in a recursion hierarchy is performed.
- uses the number of leafs *and* nodes to determine a program's granularity, not just the number of leafs, so the granularity threshold is around 0.7ms instead of 1ms. This is because only the first recursive procedure's profile is examined and there are cases, e.g. in the Barnes Hut benchmark, where this procedures does not generate any leafs at all since it calls a second recursive procedure which does the work. A more advanced version of `reapar` could just add dummy leafs for recursive branches that are not accounted for in the numbers of children in the next recursion level.
- requires that for coarse grained programs, the simulation parameters are provided by the user. Later versions of the system could derive them automatically, but e.g. even determining the branching degree affected by a Nothread annotation is not trivial.
- uses the system's timers to achieve a better granularity than the one second resolution used in the thesis's measurements.
- does not use the C compiler's `-g` option.

None of these items implicates changes in the thesis's premises, they are just technical details necessiated by compromises between a tight schedule and the will to really provide a "turn key system". A more advanced implementation of the `reapar` script would produce results identical to the ones presented in the thesis.

Chapter 4

Recursion Analysis

Both Instrumentation and Parallelization require an analysis of the source code that identifies procedures and extracts information such as

- Procedure name
- Starting and ending source code line
- Forward declarations
- Parameters used
- Return type
- Any calls made to the procedure within the program
- All calls made by this procedure

Once this information is known, recursive procedures are found by building the transitive closure of calls and checking if the procedure itself appears in it. This also takes indirectly recursive procedures into account, e.g., A calls B calls A.

In addition, during source code analysis all program annotations are detected. Annotations and their usage are described in detail in sections 3.2.2 and 3.2.3.

4.1 Algorithms Used

The following algorithms, given in pseudocode, perform the analysis. They are part of both instrumentation and parallelization.

As mentioned above, the system is based on Perl, so pattern matching and heuristics are used instead of exact parsing and construction of abstract syntax trees. This leads to some restrictions (e.g., function name and starting parenthesis of the parameter list have to be in one line, else the pattern for function recognition does not match) but allows for rapid development and makes the system independent from compiler tool availability on a given platform. The algorithms themselves are independent of the implementation, of course.

4.1.1 Procedure Identification

The algorithm in figure 4.1 identifies procedure occurrences and declarations at top level, i.e., not within any braces {}, and extracts procedure types and parameters. It also detects externally defined procedures.

```
INPUT: C source code
OUTPUT: Set of procedure names procnames, list of their starting and ending positions procstarts[]
  and procends[], list of their declarations procdecls[], list of return calls and places
  of NEEDRESULTS annotations within each procedure returns[]
bracelevel       = 0    # Number of currently open braces in line
oldbracelevel    = 0    # Ditto on line before
inside_procedure = 0    # Flags
inside_header    = 0
initialize procstarts[], procends[], procdecls[], returns[] to ∅
while (more lines)
     read next line
     recognize and remember annotations
     remove strings and enclosing quotes
     remove comments, also ones spanning multiple lines
     oldbracelevel = bracelevel
     bopens  = number of "{" in line
     bcloses = number of "}" in line
     bracelevel = bracelevel + bopens - bcloses
     if (line matches return statement)
          returns[pn] = returns[pn] ∪ current position
     end if
     if (oldbracelevel == 0 and inside_procedure == 0 and
     line matches procedure header)
          pn = procedure name
          if (pn ∈ procnames)
               mark old occurrence as declaration in procdecls[pn]
          else
               remember procedure type and parameters for pn
               procnames = procnames ∪ pn
               inside_procedure = 1
               inside_header    = 1
          end if
          procstarts[pn] = current position
     end if
     if (inside_header == 1)
          skip and remember header until
          - ";" encountered (i.e., it was just a declaration) or
          - "{" encountered (i.e., end of header)
          inside_header = 0
     end if
     if (inside_header == 0 and inside_procedure == 1 and bracelevel == 0)
          procends[pn] = current position
     end if
end while
merge return statement positions in returns[] with NEEDRESULTS annotations for later use
```

Figure 4.1: Algorithm: Procedure identification in C source code.

4.1.2 Call Identification

The algorithm in figure 4.2 identifies calls to any procedures that were detected in the earlier phase, i.e., occurrences of the procedures within one or more levels of braces {}.

```
INPUT: C Source code, lists and sets from procedure identification
OUTPUT: List of procedures called within a procedure directcalls[], list of positions
 and procedures where a procedure is called wherecalled[]
bracelevel    = 0    # Number of currently open braces in line
oldbracelevel = 0    # Ditto on line before
initialize directcalls[], wherecalled[] to ∅
while (more lines)
     read next line
     remove strings and comments, also ones spanning multiple lines
     oldbracelevel = bracelevel
     bopens  = number of "{" in line
     bcloses = number of "}" in line
     bracelevel = bracelevel + bopens - bcloses
     C = procedure containing the current line
     if ((oldbracelevel > 0 or (bopens == bcloses and bopens > 0)) and
     any known procedure P matches the "procedure call" pattern)
          directcalls[C] = directcalls[C] ∪ P
          wherecalled[P] = wherecalled[P] ∪ (C :  current position)
     end if
end while
```

Figure 4.2: Algorithm: Procedure call identification.

4.1.3 Recursion Identification

Now that all procedures and their invocations are known, a recursive procedure P is simply identified as containing P itself in its transitive closure of calls, i.e., P calls itself directly, or P calls some other procedure Q that calls R that calls ... that in turn calls P. Figure 4.3 shows the corresponding algorithm.

```
INPUT: Set of procedures procnames, list of their calls directcalls
OUTPUT: Set of recursive Procedures recursives, list of all procedures a procedure calls allcalls[]
initialize recursives, allcalls[] to ∅
for all P ∈ procnames
     sort directcalls[P] eliminating duplicates
     allcalls[P] = directcalls[P]
end for
for all P ∈ procnames
     repeat
          for all C in allcalls[P]
               allcalls[C] = allcalls[C] ∪ allcalls[P]
           end for
     until not (allcalls[P] changes)
end for
for all P ∈ procnames
     if (P in allcalls[P])
          recursives = recursives ∪ P
      end if
end for
```

Figure 4.3: Algorithm: Identification of recursive procedures.

4.2 Example

The following example output illustrates the operations performed. First, the system identifies all procedures with their respective types, parameters, starting and ending lines and source code position of their forward declaration (if any). Then, the locations of all calls to each procedure are given in the format *CallingProcedure|LineOfCall,PositionInLine:[...]*. Finally, the system prints the recursive procedure found, instruments the program code, and writes the output file:

```
> instrument_program -out fractal_instrumented.c fractal.c

Reading source code...

Found the following procedures:

  1) procedure print_pixels         from line   73 to   84
        type: void
        parameters:
           -
  2) procedure compute_pixel        from line   89 to  109
        type: unsigned char
        parameters:
           int x, int y
  3) procedure compute_frame        from line  115 to  132
        type: void
        parameters:
           int x1, int y1,
           int x2, int y2
  4) procedure compute_cross        from line  138 to  152
        type: void
        parameters:
           int x1, int y1,
           int x2, int y2
  5) procedure edges_match          from line  157 to  193
        type: char
        parameters:
           int x1, int y1,
           int x2, int y2
  6) procedure compute_area         from line  200 to  256
        declared at 68,17-68
        type: void
        parameters:
           int x1, int y1,
           int x2, int y2
  7) procedure main                 from line  259 to  334
        type: int
        parameters:
           int argc
           char **argv

Found the following procedure calls:

  1) print_pixels         called in compute_area|209,13:main|332,13:
  2) compute_pixel        called in compute_frame|121,36:compute_frame|124,36:
                                    compute_frame|127,36:compute_frame|130,36:
                                    compute_cross|144,36:compute_cross|148,36:
```

```
                                   compute_cross|150,36:compute_area|226,40:
                                   compute_area|232,40:main|317,35:
  3) compute_frame        called in main|307,14:
  4) compute_cross        called in compute_area|249,17:
  5) edges_match          called in compute_area|235,19:
  6) compute_area         called in compute_area|251,16:compute_area|252,16:
                                   compute_area|253,16:compute_area|254,16:
                                   main|309,13:
  7) main                 called in -

Found the following recursive procedures:

  1) compute_area

Writing output to file 'fractal_instrumented.c' ...
```

Chapter 5

Automatic Instrumentation

After all recursive procedures and their parameters are known, the program can be instrumented.

The goal of instrumentation is to find out at runtime how often any recursive procedure is called, what maximum recursion depth it reaches, and how often each branching degree occurs at each recursion depth. This information is recorded while the instrumented program runs and printed as the program exits.

Instrumentation is realized as the insertion of code at several places in the program source code, e.g., the addition of a parameter measuring the recursion depth for each recursive procedure.

Adding recursion tree recording code (i.e., using the `-record` or the `-time` option) works exactly the same way, so it is not explicitly mentioned below any more. The corresponding code insertions are performed during instrumentation, too.

5.1 Details

All code insertion changes are collected and then performed in order, starting at the last source code line.

Code that has to be inserted at the very beginning of a procedure is handled using a trick — a dummy variable is declared and initialized using a procedure that performs the desired initialization as a side effect, e.g.:

```
int do_our_init() {
      initialize anything we need
      return 0;
}

procedure() {
      int out_dummy = do_our_init();  /* perform initialization */
      ...
}
```

This is necessary since it is hard to detect the end of a variable declaration section in a procedure syntactically.

Profile output at the end of the program is implemented using the `atexit()` library function to ensure that profile output occurs wherever the program is exited.

To guarantee that inserted statements are executed in the same block as the code they refer to, both they and the code are encapsulated with braces. Otherwise, they might not end up in the same `if` statement as the code they are supposed to measure. Statement boundaries are recognized by looking for ";" or "{" or "}".

Thread wrappers introduced by the parallelization (see chapter 6) are recognized and ignored for instrumentation, since they already contain the depth parameter.

5.2 Algorithm Used

The pseudocode in figure 5.1 outlines the instrumentation.

```
INPUT: Source code, Information gathered during program analysis: wherecalled[R] lists all lines
  where procedure R is called, allcalls[R] lists all procedures called by R
OUTPUT: instrumented source code
Insert #include "profile.h" at the first line
Insert profile initialization code for the global profile and for each recursive procedure
at the beginning of main()
Insert atexit() handler for the final profile output
for all recursive procedures R
      Prepend recursion depth parameter profile_depth_R to the procedure's
      parameter list (in its header as well as in any declarations)
      Insert recursion branch counter branch_count to the procedure's variables
      for all return statements within R
            Insert profile information update before return:
                  ProfileAddStat(profile_stat, start line of R, profile_depth_R,
                  branch_count)
      end for
      Insert profile information update just before end of procedure
      for all calls C to the procedure in wherecalled[R]
            if (C occurs within R)
                  prepend profile_depth_R + 1 to the call's parameters
            else if (C occurs within recursive procedure P in allcalls[R])
                  prepend profile_depth_P + 1 to the call's parameters
            else
                  prepend 0 to the call's parameters, starting the recursion depth counter
            end if
            if (C occurs within recursive procedure P in allcalls[R], including R)
                  Insert branch_count++ before call
            end if
      end for
end for
perform insertion changes
```

Figure 5.1: Algorithm: Instrumenting the source code for runtime data collection.

5.3 Example

The following example shows the modifications made by the instrumentation with added characters shown in italics. 77 is the line number the instrumented procedure starts in. The profile is initialized for 100 source code lines, maximum recursion depth 40 and maximum degree 5:

```
...
void myfunction(double a,
                char *b)
{
  ...
  if (condition) {
    ...
    return;
  }
  ...
  myfunction(c, d);
  ...
  if (condition) {
    ...
    myfunction(e, f);
  }
  ...
}
...
main() {
  ...
  myfunction(a, b);
}
```

⇒

```
#include "profile.h"
...
void myfunction(int profile_depth_myfunction,
                double a, char *b )
{
  int branch_count=0;
  ...
  if (condition) {
    ...
    {ProfileAddStat(profile_stat, 77,
                    profile_depth_myfunction,
                    branch_count);
    return; }
  }
  ...
  {branch_count++;
  myfunction(profile_depth_myfunction + 1, c, d);}
  ...
  if (condition) {
    ...
    {branch_count++;
    myfunction(profile_depth_myfunction + 1, e, f);}
  }
  ...
  ProfileAddStat(profile_stat, 77,
                 profile_depth_myfunction
                 branch_count);
}
...
int do_init() {
  profile_stat = ProfileNewStats(100,40,5);
  ProfileInitLine(profile_stat,77,"myfunction");
  atexit(ProfileStatPrint);
}
main() {
  int dummy = do_init();
  ...
  myfunction(0, a, b);
}
```

Chapter 6

Automatic Parallelization

This chapter describes how code is added to allow for a parallel execution of the program's recursive branches. It then outlines the algorithm used for parallelization, and gives an example of a program before and after code insertion. Finally, the thread generation strategies employed by the parallelized program and examples of the speedups reached are detailed.

6.1 Adding Threads

To call a recursive procedure P as a thread, the system has to insert several helper procedures and data structures — e.g., a thread procedure has only one single parameter, so the procedure's parameters have to be passed to the thread wrapped into a new data structure.

The necessary additions are:

- A `struct` to hold the procedure's parameters, for passing a pointer to that struct when generating a thread for the procedure
- A wrapper procedure `ThreadWrapper_`P that unpacks the argument structure and calls the normal sequential procedure P
- A procedure `Thread_`P that takes P's parameters, packs them into the structure, generates a new thread, lets it execute P's wrapper procedure, and returns its thread ID
- A procedure `Join_`P to join the given thread ID, guaranteeing that the results computed by P are available

Also, code is inserted around each recursive call to P to make the runtime decision whether to generate a new thread and to perform the parallel or sequential call as needed. All this is summarized in figure 6.1.

Section 6.3 gives a more detailed impression of the code added for parallelization.

6.2 Algorithm Used

The pseudocode in figure 6.2 shows the Parallelization algorithm. The resulting program code is then run through the Instrumentation algorithm that supplies the necessary branch counter and recursion depths.

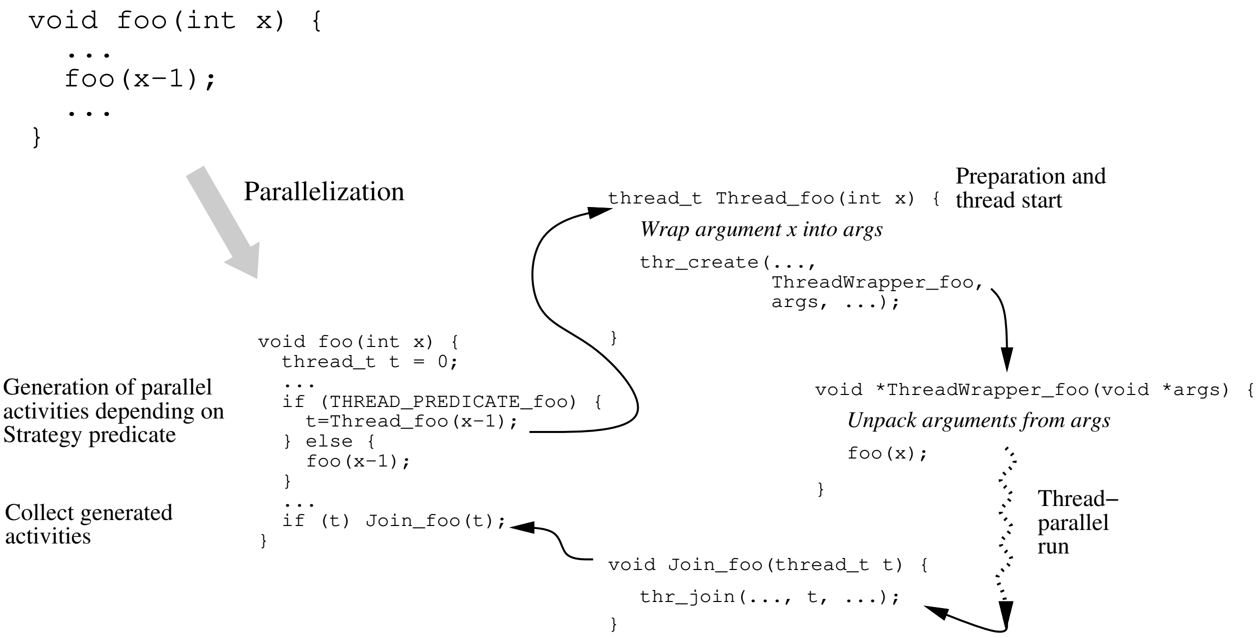


Figure 6.1: Thread generation by the automatically inserted wrapper procedures (slightly simplified, e.g. one thread variable instead of an array etc.).

6.3 Example

This example gives an impression of the infrastructure that is added for thread generation.

Take the following code (which does nothing useful at all but shows recursive branches and return statements):

```
void foo(int i) {
  if (i>100) {
    foo(i/2);
    foo(i/2+1);
    return;
  }
  if (i>10) {
    foo(i-10);
  }
}
main(int argc, char **argv) {
  foo(atoi(argv[1]));
}
```

It is transformed into the code shown in figures 6.3 and 6.4 (with abbreviated code in italics and original code in larger bold face) containing all profile instrumentation as well as definitions of thread generation strategies and support procedures for thread introduction and joining...

This code expands into around 400 lines of C source code not to be presented here — if you do want to examine the exact source, just run `parallelize_program` on the sample code above.

```
INPUT: Information from Analysis: Set of recursive procedures recursives, list of their calls
 directcalls, list of all procedures a procedure calls allcalls[]
OUTPUT: thread parallel program
insert general thread initializations and declarations at beginning of program code
topcode = initialization prefix
initcode = global thread initialization procedure template
for all recursive procedures R ∈ recursives
     threadcode = thread wrapper and join procedures template for R
     if ( (R is not annotated NOPARALLEL) and (R has return type "void") )
          add defines, struct for parameters, thread wrapper, thread generation predicates,
          and procedure declaration to topcode
          if (R has no forward declaration)
               add forward procedure declaration to topcode
          end if
          add strategy depth print statement to initcode
          for all parameter variables V of R
               add thread argument initialization for V to threadcode
          end for
          insert actual procedure name, type, arguments etc. of R into threadcode
          insert threadcode before start of R
          for all recursive procedures P ∈ recursives
               if ( (R in directcalls[P]) and (P in allcalls[R]) )
                    callcode = recursion-as-thread call code template including profile depth
                    if (P has no thread initialization code yet)
                         arrayinitcode = thread array initialization template
                         arraydecl = ""
                         joincode = thread joining code template
                         for all recursive procedures Q called in P
                              add thread array declaration for Q to arraydecl
                              add thread array initialization for Q to arrayinitcode
                         end for
                         insert arraydecl before Q's variable declaration
                         insert arrayinitcode before Q
                         for all join lines J (annotated or return) in P
                              add joincode before J
                         end for
                    end if
                    insert actual procedure name, type, arguments etc. of R into callcode
                    for all calls C of R in P
                         insert callcode around C
                    end for
               end if
          end for
     else annotate R as NOPARALLEL
     end if
end for
insert topcode at start of program
insert initcode before start of main()
insert call to initcode in main()
perform insertion changes
```

Figure 6.2: Algorithm: Introduction of thread constructs for parallelization.

```
Initialize thread definitions
Define thread generation predicates for foo
int init_threads_foo(thread_t *newthreads_foo) {
    Init thread arrays for foo
}
void foo(int profile_depth_foo, int i); /* Forward declaration */
typedef struct { /* Holds parameters for thread call */
    int profile_depth_foo;
    int i;
} ThreadArg_foo;
void *ThreadWrapper_foo(void *arg);
thread_t Thread_foo(int profile_depth_foo, int i) {
    Generate thread for foo() using ThreadWrapper_foo()
}
void Join_foo(thread_t thread_id)
    Join given thread for foo()
}
void *ThreadWrapper_foo(void *arg)
    Unpacks arguments and calls foo()
}
void foo(int profile_depth_foo, int i) {
    int branch_count = 0;
    thread_t newthreads_foo[12];
    int nt_i = init_threads_foo(&(newthreads_foo[0]));
    if (i>100) {
        branch_count++;
        if (Thread generation predicate) {
            newthreads_foo[branch_count-1] = Thread_foo(profile_depth_foo+1, i/2);
        } else {
            foo(i/2);
        }
        branch_count++;
        if (Thread generation predicate) {
            newthreads_foo[branch_count-1] = Thread_foo(profile_depth_foo+1, i/2+1);
        } else {
            foo(i/2+1);
        }
        Join threads generated using Join_foo(newthreads_foo[i])
        ProfileAddStat(profile_stat, 180, profile_depth_foo, branch_count, 1);
        return;
    }
    ...
```

Figure 6.3: Sample code after parallelization (1)

```
    ...
    if (i>10) {
        branch_count++;
        if (Thread generation predicate) {
            newthreads_foo[branch_count-1] = Thread_foo(profile_depth_foo+1, i-10);
        } else {
            foo(i-10);
        }
    }
    Join threads generated using Join_foo(newthreads_foo[i])
    ProfileAddStat(profile_stat, 180, profile_depth_foo, branch_count, 1);
}
int do_thread_init() {
    Init LWPs, set atexit() calls, print thread predicate information etc.
}
int do_profile_init() {
    Init Profile information, set atexit() calls
}
main(int argc, char **argv) {
    int dummy_profile_init = do_profile_init();
    int dummy_thread_init = do_thread_init();
    foo( 0, atoi(argv[1]));
}
```

Figure 6.4: Sample code after parallelization (2)

6.4 Parallelization Strategies

After parallelization, the program runs concurrently, generating threads in place of recursive calls.

The program's performance depends critically on good runtime decisions when to create a new thread and when to perform a sequential recursive call instead. These decisions are called "parallelization strategies". A strategy that creates too few threads leads to bad performance, since the parallelism potential is not used — creating just two threads at all on a 16-processor machine is no good idea. On the other hand, generating too many threads will swamp the machine which a lot of tiny tasks whose pure computation time may be smaller than the cost of creating and managing a thread itself, resulting in large operating system overhead for thread handling and slowing the program down.

Obviously, different programs have different characteristics that require different strategies. For coarse grained programs where one recursive call takes minutes to complete, it may even be a good decision to perform all recursive calls in parallel. On the other hand, very fine grained programs whose overall execution time is just one second and the computation performed in each recursion step is minimal are probably not worth parallelizing at all, since thread overhead is too high. Chapter 7 shows how an appropriate strategy can be automatically selected.

The REAPAR system employs the following strategies for thread generation. As mentioned, a strategy is a predicate that tells the system if it should use a new thread for the current recursive call.

General Strategies : These use the number of currently active threads as well as the overall number of threads generated. The locking necessary on these two global variables incurs some overhead; the variables do hardly represent a bottleneck, though, as the number of processors on SMP machines is small. All parameters N are multiplied with the number P of processors used, e.g., "Keep 3" on 8 processors means 24 active threads. Typical values of N lie in the range from 1 to 20.

First N: Take exactly the first $N \times P$ opportunities to generate a thread. Given the Solaris scheduling strategy, this corresponds to the parallelization of the upper calls in the recursion tree.

Keep N: Generate a thread iff the number of currently active threads is less than $N \times P$, distributing work over the tree of recursive calls.

Active N: Same as Keep N, but the thread counter is already decremented after the procedure's call within the thread returns, not after the thread is joined, thus allowing for a larger number of concurrent computations but increasing the number of unjoined threads.

Always: Start a thread whenever it is possible (Keep ∞). This is useful for coarse grained problems.

Never: Do not start any threads at all (Keep 0), which is sensible if the thread creation cost is higher than the cost of the computation itself.

Neverever: Like Never, but even without thread constructs and thus less overhead.

Problem-specific Strategies : Make use of available program-specific parameters (currently only the recursion depth). The range of values considered can be read from the runtime profile.

Depth D: Generate threads in the first D levels of the recursive call tree but not below that. This strategy avoids creating threads for small subproblems near the leafs that are better solved sequentially. Since no thread count is needed here, there is no locking overhead, so this strategy is better suited for fine-grained programs.

Combined Strategies : Combine general and problem-specific parameters.

Keep N until Depth D: Create threads if both the current depth is less than D and the current number of threads is less than $N \times P$. Reduces the number of threads generated while avoiding small subproblems, but uses locking.

Active N until Depth D: Ditto using the Active N strategy.

The default strategy used by the system after Parallelization (without Strategy Selection) is Active 3, which is a good general-purpose strategy. For fine-grained programs, a Depth strategy is more appropriate. Combined strategies are conceptually elegant for several problems, but no combined strategy yielded better speedups than a General or Depth strategy for the REAPAR benchmarks, so combined strategies are given no further consideration.

6.4.1 Two Examples

Figure 6.5 shows the speedup and user/system time measured for the strategies Neverever, Never, Always, First, Keep, Active and Depth, each with parameters ranging from 1...7 (First) resp. 1...20. The benchmark used is the coarse grained Eigenvalue benchmark with geometrical input distribution which results in an unbalanced recursion tree. Therefore, both Keep and Active strategies reach a perfect speedup while the Depth strategy performs worse. In this case, a high enough Depth parameter would do the job, too, as the good performance of the Always strategy shows.

On the other hand, figure 6.6 shows the corresponding curves for the very fine grained Bitonic Sort benchmark. Any parallelization strategy class but Depth performs very badly since the locking overhead of the General strategies is too high in comparison to the small computation involved. This is also evident from the system time (i.e. thread overhead) that increases dramatically with the Keep and especially Active strategies. From Depth 13 on, the Depth strategy generates more unjoined threads than Solaris can handle (around 3 000), resulting in aborted runs which are shown with speedup 0.

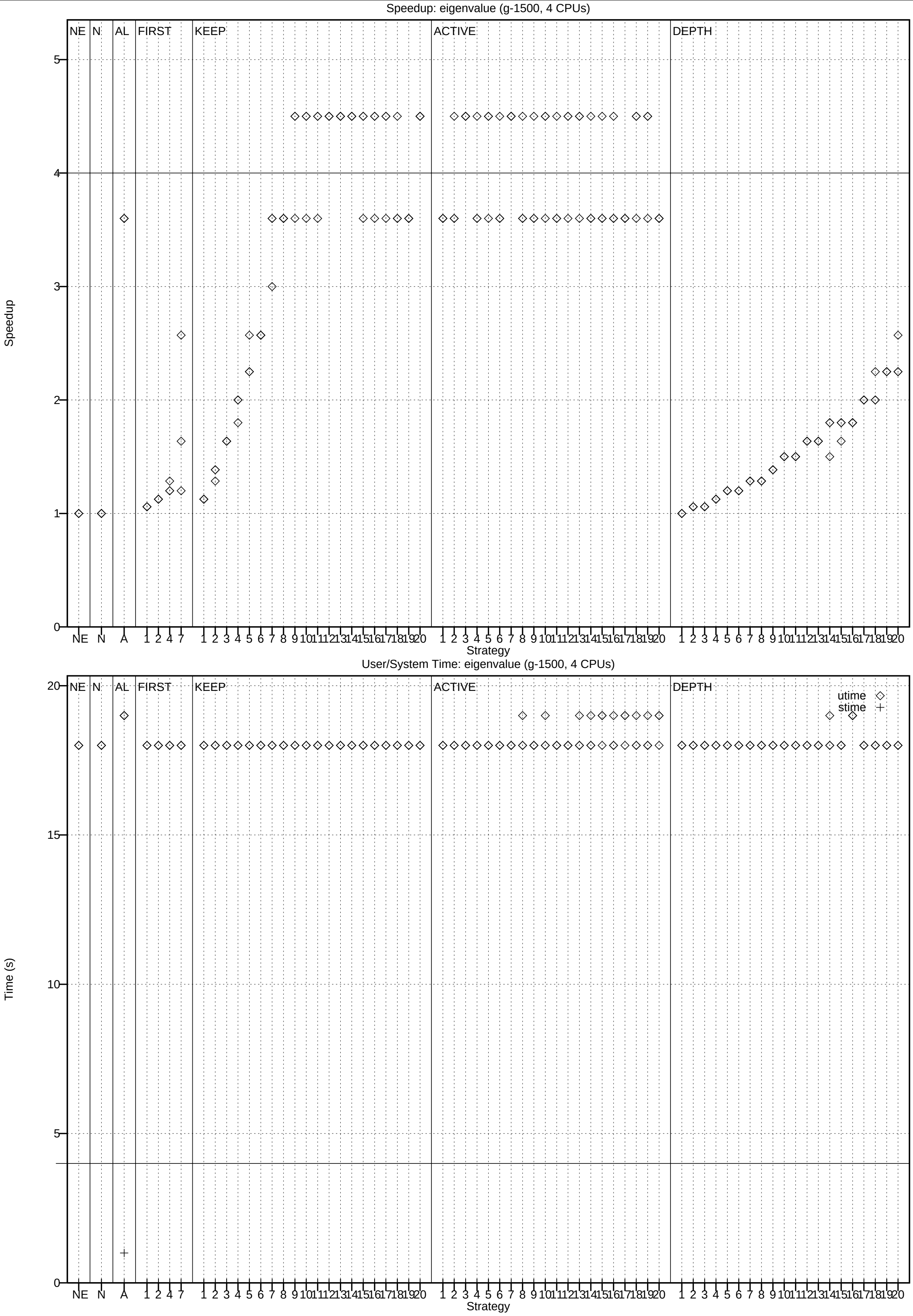


Figure 6.5: Speedups and cumulative user/system time on all CPUs for the Eigenvalue benchmark, input size "g 1500".

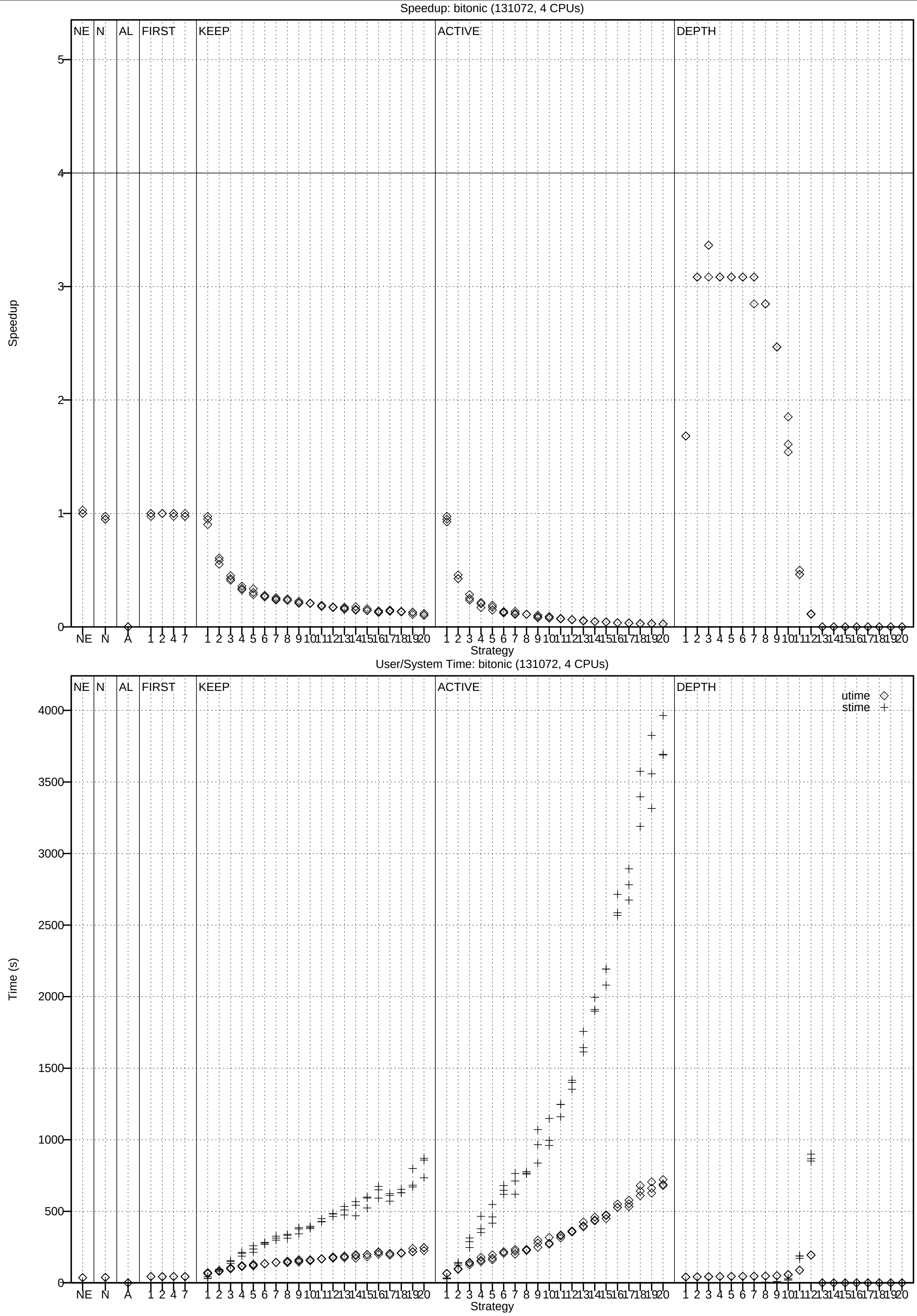


Figure 6.6: Speedups and cumulative user/system time on all CPUs for the Bitonic Sort benchmark, input size 131 072.

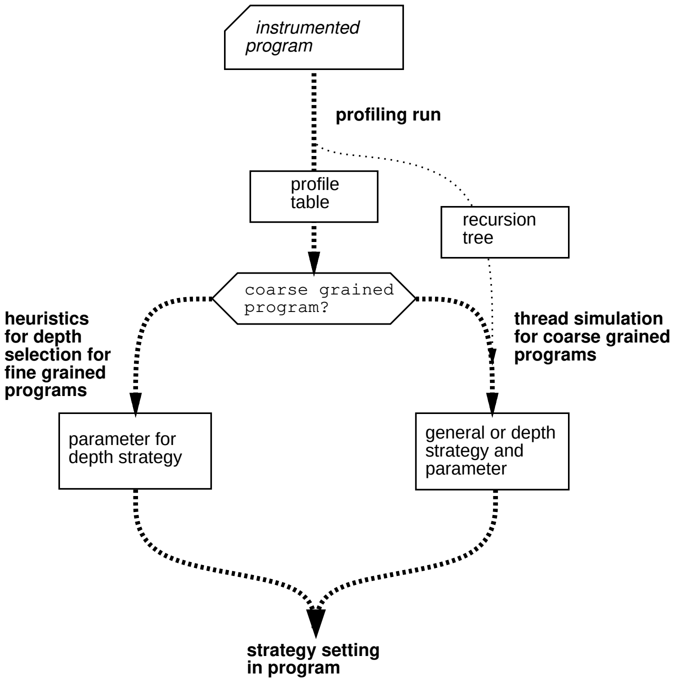
Chapter 7

Automatic Selection of Parallelization Strategies

This chapter explains how the automatic strategy selection works and what the considerations behind it are.

7.1 Coarse and fine grained programs

A suitable parallelization strategy for a given program and input data set is chosen as shown in the following picture:



The program is classified as either coarse grained or fine grained according to the average time it takes to compute a leaf of the program's recursion tree. Fine grained programs only reach good speedups with a Depth strategy, so for them a good strategy parameter is chosen according to the program's profile table. For coarse grained programs, the recursion tree is recorded and a thread simulation is performed for all reasonably possible parallelization strategies and parameters and the best performing strategy is chosen.

With the 100MHz HyperSparc system REAPAR was developed on, any problem whose leafs take less than one millisecond are considered fine grained. On other systems, this threshold has to be set empirically once at system installation time.

7.2 Profile evaluation

For fine grained programs, REAPAR evaluates the profile resulting from the instrumentation.

7.2.1 Heuristics

Using the Depth strategy with parameter D, the number of threads generated is equal to the number of nodes and leafs of all parallelized procedures in the recursion tree[1]. Therefore, we can derive a good value for D on C CPUs if we estimate the maximum size of a subtree at depth D and then apply the following heuristics:

- The largest sequentially executed subtree must be smaller than $1/C$ of the complete tree. Otherwise, the problem cannot be divided equally among the C processors.
- The number of nodes up to and including depth D must be at least C to offer enough parallelism.
- The number of threads generated (i.e., nodes and leafs up to D) must be smaller than 3 000, since Solaris only manages about that many unjoined threads and aborts if more are created.

7.2.2 Subtree size estimation algorithms

There are two ways of estimating the size of a subtree:

LPS (Largest Possible Subtree): The LPS(D) describes how many nodes/leafs can be contained at maximum in a subtree starting at depth D. It is derived from the profile table by starting at depth D and picking a node with highest possible branching degree m, then examining depth $D+1$ and picking m nodes with highest possible branching degree, then picking the corresponding number of nodes from depth $D+2$ and so on. Its size, i.e., the number of nodes `LPSnodes` in the LPS, is implictily constructed by the following algorithm, where $p(D,g)$ is the number of branches of degree g, D_{max} is the maximum recursion depth, and g_{max} the maximum branching degree:

```
LPSnodes = 1; getnodes = 1
for d = D to D_max
   getnextnodes = 0 /* How many nodes to get on the next layer */
   /* Get maximum number of nodes with maximum branching degree */
   for g = G_max downto 0
      n = min(getnodes, p(d,g))
      getnodes = getnodes - n
      LPSnodes = LPSnodes + n
      getnextnodes = getnextnodes + g*n
   end for
   getnodes = getnextnodes
end for
```

[1] If a Nothread annotation is in effect, there are less threads generated, but for the worst case estimate used here, this does not make much of a difference.

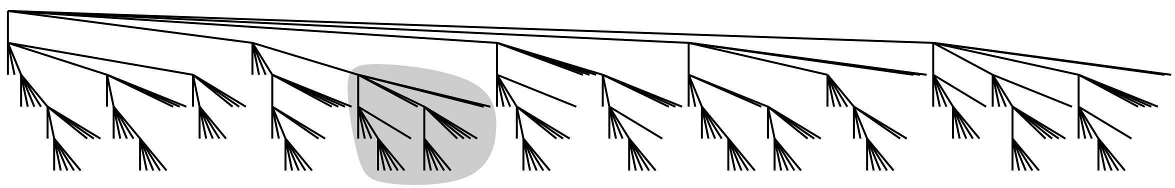
After the loop, the percentage of nodes in the LPS relative to the complete tree can be compared to $1/C$.

AS (Average Subtree): The AS describes how large a tree at depth D is on average, i.e., always using the average branching degree instead of the maximum degree available as in the LPS:

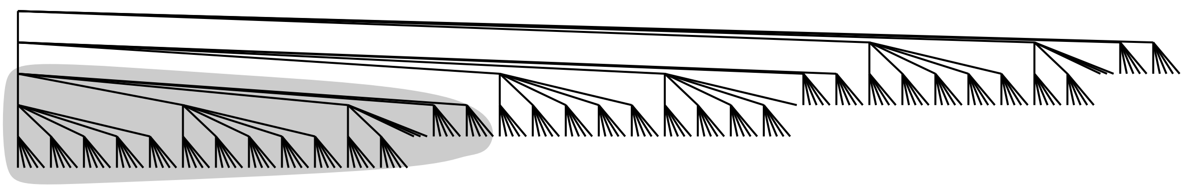
```
ASnodes = 1; getnodes = 1
for d = D to D_max
    /* Compute average branching degree, weighted by number of nodes */
    layernodes = 0
    avgdegree = 0
    for v = V_max downto 1
        avgdegree = avgdegree + g * p(d, g)
        layernodes = layernodes + p(d, g)
    end for
    avgdegree = avgdegree / layernodes
    n = min(getnodes, layernodes)
    nodes = nodes + n
    /* Calculate average number of nodes to get in next layer */
    getnodes = avgdegree * n
end for
```

To visualize these subtrees, the following picture shows an actual recursion tree with a typical subtree highlighted, compared to the corresponding LPS. The tree containing the LPS was derived from the original tree by the LPS algorithm (maximim branching degree first).

Actual recursion tree with typical subtree at depth 2



LPS−conforming tree generated from actual tree, with corresponding LPS at depth 2



7.2.3 Example and prerequisites

Profile evaluation is performed by the program `evaluate_profile.c` which reads a given profile and analyzes it for each depth. The following sample output gives a further impression of the program, running on a profile of the Fractal benchmark for 5 CPUs:

```
profiled procedure 'compute_area' at source line 226
maximum degree = 4, maximum depth = 9
profile covers 1 iterations
recommend using the NOTHREAD annotation for branching degree 4
 since degree is always 0 or 4.
performing Average Subtree analysis for 5 CPUs

analyzing depth 1:

all in all, the average subtree has 17178.0 nodes.
that's  25.00% of the tree's nodes
-> not recommended for 5 CPUs, average subtree not smaller than 1/5 (20.00%)

analyzing depth 2:

all in all, the average subtree has 4295.0 nodes.
that's   6.25% of the tree's nodes
-> RECOMMENDED: depth  2 for  5 CPUs, average subtree is only 6.25%
```

One level above this evaluation, the wrapper script `choose_strategy_by_heuristics` takes a profile and a cpu number, calls the evaulation, and outputs the first depth recommended resp. the fact that no depth could be recommended.

The evaluation also works for profiles that contain several iterations over a recursive procedure, i.e., whose first line has more than one entry. In this case, each entry in the profile is divided by the number of iterations before analysis. By default, AS analyis is used since the LPS heuristic tends to be too pessimistic and recommends higher depths than the AS heuristic.

Evaluation always takes the first profile in the profile output as source for its data. You have to make sure the first profile is the profile of the procedure you want to parallelize. More than one procedure can be taken into account by merging profiles, thus creating a larger profile that is the sum of both procedures' profiles. You can manually force profile merging by removing the `"[...]"` separator between profile tables in the profile output and using the edited output as input for the analysis.

7.3 Thread simulation

For coarse grained programs, the whole recursion tree can be used to simulate the progress of virtual threads on virtual CPUs working on the recursion tree.

7.3.1 Abstractions and measures

The simulation relies on the following abstractions:

- Executing one node or leaf takes a single time step.
- Generating, switching, joining and terminating threads occur at once in zero zime steps.

- A processor works on a thread until the thread is out of work or waits for its children. No thread switch by process time slice expiry is simulated.
- Since the tree shows no points where computation was joined, a node's thread waits for the return of each child thread after the node's work has been done.

The result of a simulation is the recursion tree with annotations about which thread executed which node and the number of simulation steps. Further statistics about the maximum number of nodes a single thread worked on (as a measure of sequentiality) and the simulated load on the CPUs can be derived easily from this.

7.3.2 Algorithm used

A simulated thread T in the array of threads `threads[]` is in one of the states NONE (inactive), ACTIVEly computing, WAITING for child threads, DONE with its computation or JOINED by its parent thread. The simulated thread contains data on its state, the number of nodes/leafs it has worked on, the number of non-parallel nodes/leafs it has worked on, its starting and maximum depth, its start node and its current node.

A simulated CPU C has a thread running on it (array `cputhreads[`C`]`) or is marked "idle". Non-idle CPUs contribute to the simulated load of the machine.

Nodes n in the tree know about their children, their parent node, the number of nodes in their subtree, the procedure that generated them, the thread working on them and their depth in the tree. Additionally, boolean variables show if the nodes is DONE and if all its subnodes were generated by procedures that were not parallelized, i.e., if the node is a "parallel leaf".

Furthermode, for all procedures P that built up the recursion tree, information is kept stating if they were parallelized, if there was a Nothread annotation (and its degree), how many nodes they generated and how many non-parallel subnodes they generated indirectly.

Based upon this background, figure 7.1 shows the algorithm of the simulation without going into details such as activity counters or leaf counters.

After simulation, the suitability of the parallelization strategy used for C CPUs is evaluated by the following heuristics:

- At least C threads were generated.
- The percentage of nodes worked on by the largest thread is less than $1/C$ of the complete tree.
- The same has to be valid for the number of nodes and leafs from sequential procedures (e.g., if just the upper recursion layer was parallelized).
- Each simulated CPU is busy at least 75% of the time.

7.3.3 Example

The simulation is performed by the `simulation.c` program. In the following output shows we see its results for the Keep 4 strategy on 4 CPUs based on the Barnes Hut program's recursion tree:

```
INPUT: Recursion tree and parameters to the simulation
OUTPUT: Recursion tree, nodes marked with the thread number that executed them, and thread statistics
  such as the size of the largest thread
read the recursion tree and parse it, noting which procedures executed which nodes
parse command line parameters, e.g. strategy and number of processors
initialize the virtual threads and the thread queue
for all C in CPUs
     cputhreads[C] = NONE
end for
thread[0].node = rootnode; cputhreads[0] = 0; simsteps = 0
while rootnode.state != DONE                          Simulate until whole tree done
     for all C in CPUs
          childrendone = False                        Are all the node's children done?
          T = cputhreads[C]                           Get current thread for this CPU
          if T is NONE
               T = dequeue_thread(C)                  CPU has no current thread, dequeue one
               if T is NONE then
                    Mark CPU idle and skip to next CPU
               end if
          end if
          N = T.node
          if all of N's children are DONE, JOINing them while checking
               N.state = DONE                         All children done → node done
               if T.startnode == N                    At thread's start node? Thread done!
                    cputhreads[C] = NONE              Remove thread from CPU
               T.state = DONE; break for
               else                                   Otherwise: Proceed upwards in tree
                    T.state = ACTIVE
                    T.node = N.parent
               end if
               childrendone = True
          end if
          switch (T.state)
          case WAITING:
               if not childrendone                    Still waiting for children, thread sleeps
                    cputhreads[C] = NONE              Remove thread from CPU...
                    enqueue_thread(T)                 ...and put it in Q for later use
               end if
          case ACTIVE:
               if not childrendone                    Children left to work on?
                    if all of N's children are busy
                         T.state = WAITING
                    else                              Start work on child as new thread...
                         if parallelization strategy says so
                              spawn_and_enqueue_new_thread(N.child)
                         else                         ...or compute child sequentially
                              T.node = N.child
                         end if
                    end if
               end if
          end switch
     end for
     simsteps++
end while
```

Figure 7.1: Algorithm: Simulation of thread process in the recursion tree.

```
> simulation -k 4 -c 4 < recorded/recorded_barnes_8
  Thread   0: start   0, max   0,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   1: start   1, max   5,   4 nodes,   8 leafs,   4 subnodes,   8 subleafs
  Thread   2: start   2, max   2,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   3: start   3, max   3,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  [...]
  Thread  22: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  23: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs

Simulation for KEEP 4 on 4 CPUs done. 23 threads generated over all
57 simulation steps elapsed, CPU utilization = 339%
All procedures simulated as parallelized

Procedure B:    10 nodes,     0 leafs,    10 parnodes,     0 parleafs
Procedure C:    16 nodes,    64 leafs,    16 parnodes,    64 parleafs

Thread statistics for 24 threads:
 nodes:        0 min,     1 med,     4 max,     1.08 avg,    1.256 dev ( 1.159)
 leafs:        0 min,     1 med,     8 max,     2.67 avg,    2.953 dev ( 1.108)
 subnodes:     0 min,     1 med,     4 max,     1.08 avg,    1.256 dev ( 1.159)
 subleafs:     0 min,     1 med,     8 max,     2.67 avg,    2.953 dev ( 1.108)
Largest values:     4 nodes    ( 15.38%),   8 leafs    ( 12.50%),
                    4 subnodes ( 15.38%),   8 subleafs ( 12.50%)
Heuristics:
Not less threads than CPUs                        : OK
Node percentage of largest <= 1 CPU percentage    : OK
Leaf percentage of largest <= 1 CPU percentage    : OK
Subnode percentage of largest <= 1 CPU percentage : OK
Subleaf percentage of largest <= 1 CPU percentage : OK
Each CPU used at least 75% of the time            : OK
```

The simulation program itself takes the following options:

```
simulation - simulate effects of thread generation strategies for
     recursion tree read from stdin
Program options:
    -s A        Simulate always strategy (also '-A')
    -s a [n]    Simulate active_n strategy (also '-a [n]')
    -s d [n]    Simulate depth_n strategy (also '-d [n]')
    -s f [n]    Simulate first_n strategy (also '-f [n]')
    -s k [n]    Simulate keep_n strategy (also '-k [n]', default)
    -s n        Simulate never strategy (also '-n')
    -c C        Simulate C CPUs (default = 4)
    -p N        Strategy parameter: N*C threads with keep/active
                resp. depth N (default = 3)
    -t ABCD..   Only execute procedure type A/B/... in parallel
                (A=1st recursive procedure etc, default=all)
    -b Ax       Don't create thread for branch #x of proc. A
                (A as above, branches count starting from 0)
    -e  Execute non-parallel parts in one single step
    -q  Quiet output, don't print final tree with threads
    -v  Verbose output
Output format for one node: PDTTTS
    P   = Node's procedure (A,B,...)
    D   = Node's state ('.' incomplete / 'd' done)
    TTT = Thread ID working on that node ('===' -> none)
    S   = Thread's status (None, Active, Waiting, Done, Joined)
```

7.3.4 Strategy selection

A single simulation just tells us how well a single strategy will perform. To select the best strategy, the wrapper script `choose_strategy_by_simulation` starts the simulation with each reasonable strategy and parameter (Keep, Active, and Depth with parameter values 1…20) and chooses the strategy that takes the minimum number of simulation steps. This is a very good indicator of the strategy that performs best in reality, as the scientific work [Hän98] shows.

Since the simulation runs just a few seconds even for large recursion trees, it is much faster than actually executing the program with each parallelization strategy and finding the optimum strategy by trial and error.

An example output of the strategy selection script for the Eigenvalue benchmark follows:

```
> choose_strategy_by_simulation 8 recorded_eigenvalue_g_1500  '-b A1'

Overall ranking:
Rank 1: Active  3  (607 steps)
Rank 2: Keep   17  (634 steps)
Rank 3: [Depth not recommended]
```

The final parameter gets passed to the simulation and tells it that a Nothread annotation for the second branch of procedure A is in effect. Active 3 ranks first, followed by Keep 17. No depth strategy is recommended since the geometric Eigenvalue recursion tree is very imbalanced and performs bad with Depth strategies.

As with the profile, the user has to make sure that the first recursion tree in the program's output is the one to be parallelized. In the case of several iterations, only the first tree is taken into account.

7.4 `reapar` Script

The script `reapar` performs the automatic strategy selection and parallelization as shown in the diagram at the beginning of this chapter. Its underlying algorithm is shown in figure 7.2.

The script's limitations and ways to handle them are described in detail in section 3.3. Its current implementation is a prototype that handles all the benchmarks used for the design and validation of the REAPAR system and should have no problems with programs containing just one recursive procedure. This proof-of-concept script demonstrates that the REAPAR methods fulfil their promises and do indeed parallelize C programs fully automatically.

An industrial strength implementation of the `reapar` script could overcome the restrictions mentioned, e.g. automatically supply information about annotations to the simulation component, but this would exceed the time frame for a PhD work. The existing system shows that the concepts employed are powerful enough to support the nontrivial range of applications covered by the benchmarks, and the validation set of benchmarks proved that the methods are general enough to handle completely new programs. Thus, the current system is a solid base for further developments.

parse and interpret program parameters, check if the input can be read etc.
instrument the source code for profile generation using `instrument_program`
compile the resulting source code
run the resulting executable using the parameters given
analyze the instrumtation's output
`if` *granularity is fine* `then`
    choose a Depth parallelization strategy parameter using `choose_strategy_by_heuristics`
`else`
    instrument the source code for recursion tree recording using `instrument_program`
    compile the resulting source code
    run the resulting executable using the parameters given
    choose a parallelization strategy and parameter using `choose_strategy_by_simulation`
`fi`
parallelize the source code using `parallelize_program`
set the chosen parallelization strategy and parameter using `set_strategy_parameters`
compile the resulting source code, yielding the final parallel executable

Figure 7.2: Algorithm: Automatic strategy selection and parallelization in the `reapar` script.

Chapter 8

Summary: REAPAR Results

This brief chapter summarizes the results of the REAPAR research without going into details. A closer discussion can be found in the corresponding PhD thesis [Hän98].

8.1 Speedups in comparison to related systems

REAPAR yields the following speedups on 4-processor benchmark runs, compared to the related Cilk [Sup97] and Olden [Car96] systems:

Benchmark	Barnes Hut	Bitonic Sort *	Eigenvalue	Fractal	Heat *	Knapsack *	Magic *	Power	Queens
Olden	3,0	2,3	—	—	—	—	—	3,8	—
Cilk	3,1	—	—	—	3,9	3,6	—	—	3,9
REAPAR	3,4	3,4	4,0	3,8	4,2	3,1	3,7	3,6	3,9

Benchmarks marked as "*" are part of the validation set which was only examined after the work on the REAPAR system was completed. The speedups prove that the system was not fine tuned for the benchmarks used in its construction. Thus, the methods developed are generally useful.

8.2 Quality of automatic strategy selection

The following table shows how large a percentage of the optimum parallelization strategy's speedup the automatically chosen strategy typically obtains:

Benchmark	Barnes Hut	Bitonic Sort	Eigenvalue	Fractal	Heat	Knapsack	Magic	Power	Queens
Performance	100%	100%	100%	100%	95%	90%	100%	100%	100%

REAPAR chooses a parallelization strategy that performs best in reality, too, with just two exceptions that still obtain at least 90% of the optimum.

8.3 Speedups on more processors

After the REAPAR work was finished, an opportunity arose to measure the speedups of several programs parallelized by REAPAR on machines with 8, 12 and 30 processors[1]. The table lists the speedups reached and the strategy selection's quality:

Benchmark	Barnes Hut	Eigenvalue	Fractal	Power	Queens
Speedup 8 CPUs	5,0	8,3	7,3	7,3	4,2
Strategy performance	100%	100%	73%	—	100%
Speedup 12 CPUs	5,9	16,5	10,0	7,3	3,4
Strategy performance	100%	100%	50%	—	100%
Speedup 30 CPUs	8,2	33,0	20,0	9,8	2,8
Strategy performance	11%	100%	100%	—	32%

The excellent Eigenvalue results show that the speedup of coarse grained programs can scale perfectly. Despite problems with too short run times (Fractal), too little parallelism potential (Power), and too fine granularity (Queens), most of these results are more than adequate and underline that REAPARs methods are valid also for much higher numbers of processors than four CPUs.

[1]Many thanks to P.Hausdorf and U.Graef at the Sun Benchmark Center Germany, to P.Möller, and F.Haberhauer at sun and to H.Herrmann for his successful lobbying!

Bibliography

[BH86] Joshua Edward Barnes and Piet Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446, 1986.

[Car96] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[CRY94] Soumen Chakrabarti, Abhiram Ranade, and Katherine Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, USA, 1994. IEEE.

[Hän98] Stefan U. Hänßgen. *Effiziente parallel Ausführung irregulärer rekursiver Programme.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, April 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (ANSI C).* Prentice Hall, 2nd edition, 1988.

[LML+95] Steven S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. Technical report, Department of Computer Science, UCSB, Berkeley, 1995.

[Sun94] SunSoft, Mountain View, CA, USA. *Solaris 2.4 Multithreaded Programming Guide*, August 1994.

[Sup97] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MS, USA. *Cilk 5.0 (Beta 1) Reference Manual*, March 1997.

[WC97] Larry Wall and Tom Christiansen. *The Perl Language Home Page.* `http://www.perl.com/perl/index.html`, 1997.

[Wol96] Michael J. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, Redwood City, CA, USA, 1996.