A Case Study of Semi-Automatic Parallelization of Divide and Conquer Algorithms Using Invasive Interactive Parallelization

Erik Hansson

Reg Nr: LIU-IDA/LITH-EX-A–09/029–SE
Linköping 2009
Examensarbete

A Case Study of Semi-Automatic Parallelization of Divide and Conquer Algorithms Using Invasive Interactive Parallelization

Erik Hansson

Reg Nr: LIU-IDa/LITH-EX-A-09/029-SE
Linköping 2009

Handledare: M. Chalabine
IDA, Linköpings universitet

Examinator: C. Kessler
IDA, Linköpings universitet
Since computers supporting parallel execution have become more and more common the last years, especially on the consumer market, the need for methods and tools for parallelizing existing sequential programs has highly increased. Today there exist different methods of achieving this, in a more or less user friendly way. We have looked at one method, Invasive Interactive Parallelization (IIP), on a special problem area, divide and conquer algorithms, and performed a case study. This case study shows that by using IIP, sequential programs can be parallelized both for shared and distributed memory machines. We have focused on parallelizing Quick Sort for OpenMP and MPI environment using a tool, Reuseware, which is based on the concepts of Invasive Software Composition.
Abstract

Since computers supporting parallel execution have become more and more common the last years, especially on the consumer market, the need for methods and tools for parallelizing existing sequential programs has highly increased. Today there exist different methods of achieving this, in a more or less user friendly way. We have looked at one method, Invasive Interactive Parallelization (IIP), on a special problem area, divide and conquer algorithms, and performed a case study. This case study shows that by using IIP, sequential programs can be parallelized both for shared and distributed memory machines. We have focused on parallelizing Quick Sort for OpenMP and MPI environment using a tool, Reuseware, which is based on the concepts of Invasive Software Composition.
Acknowledgments

First of all I want to thank my supervisor Mikhail Chalabine, for his enthusiastic encouragement of the work as well as his help concerning both practical and theoretical problems. Secondly I want to thank my examiner Christoph Kessler for his feedback, guidance and interesting comments. Then I want to thank my opponent and friend Dan Persson. I also want to thank everyone at PELAB. Last I want to thank my friends and family for their support.
Contents

1 Introduction .......................................................... 1
  1.1 Task ........................................................................... 1
    1.1.1 Motivation ......................................................... 1
    1.1.2 Problem formulation ............................................. 2
    1.1.3 Requirements and preconditions .............................. 2
    1.1.4 Validation principle ............................................. 3
  1.2 Outline ................................................................. 3

2 Background ............................................................... 5
  2.1 High performance computing and parallelization ............. 5
    2.1.1 High Performance Computing ................................. 5
    2.1.2 The problem of parallelization ............................... 5
    2.1.3 Parallelizing for shared memory machines using OpenMP . 7
    2.1.4 Parallelizing for distributed memory machines using MPI . 7
  2.2 Component software ................................................ 8
    2.2.1 Levels of composition ........................................... 8
    2.2.2 Component systems ............................................. 8
  2.3 Invasive Software Composition .................................... 9
  2.4 Reuseware ............................................................. 10
  2.5 Invasive Interactive Parallelization ............................... 12
  2.6 Divide and conquer .................................................. 12
  2.7 Quick Sort .............................................................. 13

3 Implementation ......................................................... 15
  3.1 Manual parallelization .............................................. 15
    3.1.1 OpenMP .......................................................... 15
    3.1.2 MPI ............................................................... 17
  3.2 The composers ......................................................... 18
    3.2.1 OpenMP .......................................................... 19
    3.2.2 MPI ............................................................... 21
    3.2.3 Implementing in Reuseware –personal reflections ......... 24
4 Results and discussion 27
  4.1 Results ........................................ 27
    4.1.1 Speedup .................................... 27
    4.1.2 Correctness .................................. 28
    4.1.3 Fulfilling the requirements ................. 30
    4.1.4 Fulfilling the hypothesis ..................... 30
  4.2 Related work ...................................... 31
  4.3 Future work ....................................... 34

Bibliography 35

A Source Code 39
  A.1 qs_transform2.bc .................................. 39
  A.2 mpi-transform.bc ................................ 42
  A.3 mpi-init.bc ...................................... 47
  A.4 mpi-finalize.bc .................................. 48
  A.5 mpi-print-with-rank.bc .......................... 48
  A.6 mpi-init-array.bc ................................. 49
  A.7 mpi-print-array.bc ............................... 49
Chapter 1

Introduction

This master thesis project was done at PELAB, department of computer science, Linköping University. In the first chapter we will go through the task and motivation for it as well as problem formulation with requirements. We will also state a hypothesis that we in the last chapter motivate why it is fulfilled.

1.1 Task

The overall goal of this work is to perform a case study of (semi) automatically parallelizing serial code that concerns a divide and conquer problem (see Section 2.6). The general concept we will use is Invasive Software Composition (ISC), for more information see Section 2.3. In this case study we will particularly use Invasive Interactive Parallelization (IIP), see Section 2.5.

The focus will be to parallelize a serial Quick Sort algorithm, see Section 2.7, which is an instance of a divide and conquer algorithm. The result should be parallelized code for both shared-memory machines using OpenMP, see Section 2.1.3, and Message Passing Interface (MPI), see Section 2.1.4, for distributed memory machines.

1.1.1 Motivation

The main motivation for this case study is that computers with capability of parallel execution are becoming more and more widespread, not only for computers used in the scientific area but also in off-the-shelf computers [1]. Of course, parallelizing a program can not be done for free. One problematic issue with parallel computing, as everyone knows who tried it in practice, is that even the smallest parallelization problem requires a lot time both for analyzing the problem and doing the actual programming. By separating the serial core and the parallelization concern we think that the parallelization process can be made more user friendly and by using IIP we will increase the level of reusability. The motivation for studying parallelization of the Quick Sort algorithm is that the algorithm is widely used
and could be classified as a divide and conquer problem. If we manage to parallelize the Quick Sort algorithm, it is a first step towards parallelizing other divide and conquer problems.

1.1.2 Problem formulation

The problem can be formulated as: Is it possible to parallelize the Quick Sort algorithm (semi) automatically, by using IIP? By showing this we will prove the following hypothesis.

Hypothesis

Even though software composition methods such as ISC and IIP only have been evaluated for parallelizing problems with well structured parallelism, these methods are also applicable on instances of divide and conquer problems in such way that the result will give acceptable speedup figures for both shared and distributed memory machines. The only example we know of is the preceding work [3] concerning Gaussian elimination.

1.1.3 Requirements and preconditions

This work will mainly be based on a previous work [3] where some ideas and concepts in IIP already have been practically proven including development. Continuing that work will automatically give the following preconditions:

- The paradigm should be IIP.
- The software should be developed using the Reuseware [10] plug-in for Eclipse [25].
- The language of serial and generated parallel code should be C.
- Frameworks for parallelization should be OpenMP and MPI.

The given preconditions will more or less define the requirements. The task should be solved so it also fulfills the following requirements:

1. It should be possible to manually refactor the syntax tree to insert parallelization code. (priority A)

2. Generated parallel code should give the same result as the serial one. (priority A)

3. Parallelization should also be done automatically with pattern matching. (priority B)

4. Two versions of parallelized code, one for OpenMP and one for MPI (priority A)
1.1.4 Validation principle

If we manage to create the two transformed parallel versions of Quick Sort we need to validate that they actually work. Since this is a case study we will run our transformed Quick Sort algorithms to check both that we get acceptable speed up, and that the algorithms actually do correct sorting.

1.2 Outline

Here follows the outline of the thesis, the outline is based on how the work was proceeding.

Chapter 2 covers the background needed for solving the task, and is the result of a literature recherche containing a brief overview of the areas.

Chapter 3 describes the implementation steps we did for solving the task; manual parallelization for both shared and distributed memory machines, implementation of the composer used to parallelize the code semi-automatically, and also a discussion of what practical problems we ran into during the implementation.

Chapter 4 describes the result of our work, gives a comparison to already existing work and what we think will be suitable for future work. In the result section we show that we got acceptable speedup, we also motivate why our hypothesis holds.
Chapter 2

Background

Since the case study that we will perform touches many different areas in computer science, this chapter will cover a lot of different subjects that at a first glance look really far from each other. We think they are important to mention for giving the right understanding. This chapter is mainly based on a literature recherche of existing work.

2.1 High performance computing and parallelization

Using large supercomputers, usually clusters of servers connected via a high speed network, is called High Performance Computing (HPC). Even though HPC and parallelization go hand in hand, parallelization can be applied on applications used for office and home due to the fact that modern PCs usually have a multi-core processor architecture.

2.1.1 High Performance Computing

In this work we will consider two types of high performance computers, shared memory and distributed memory computers. These two types have different programming styles, which add an extra parameter to the parallelization problem in general; usually you must know in advance what type of computer you will write your parallel program for. Processors in a shared memory machine share the same view of the memory, which means that each processor can access each memory location. In contrast, a distributed memory machine doesn’t share any memory between the processors, instead all exchange of data is done by message passing and the sending and receiving has to be done by the program itself [7].

2.1.2 The problem of parallelization

To gain any performance when running a program on multicore machine or a multiprocessor system sequential code has to be changed, this is called parallelization.
A main criterion when parallelizing is that the parallel version should execute faster than the serial. It is also wanted that the program should execute \( p \) times faster if the program is run on \( p \) processors, which quite seldom is the case. There are basically four ways of parallelizing serial code, which are more or less common today.

1. **Manual refactoring:** This is the most widely spread approach and allows the programmer to change the serial code in any way to make it parallel. This is considered complex and time consuming, but a highly skilled programmer can utilize the hardware efficiently to gain good performance [1].

2. **Automatic parallelization:** Is done by the compiler and is convenient to use since the programmer does not have to change the serial code. Due to the static analysis of the code, not all potential parallelism can be utilized [1].

3. **Skeletons:** Skeletons are blackbox components with well defined interfaces that are parameterizable. Skeletons are considered to have two drawbacks: the serial code needs to be heavily refactored, and only well structured parallelism can be utilized [1].

4. **Interactive parallelization tools:** This kind of tools combine the knowledge of a parallelization expert and the help of a parallelizing compiler. For example when the compiler itself can not determine what action to do, the parallelization expert can give directives to solve the problem [1].

When discussing how "good" a parallel algorithm is the metric "speedup" is often used. Speedup can be defined in several ways, two common definitions are relative and absolute speedup[16].

Relative speedup is defined as the execution time of the parallel algorithm using one processor (core), \( T_{\text{par}}(n, 1) \), divided by the execution time of the same algorithm using \( p \) processors, \( T_{\text{par}}(n, p) \), when running the algorithm on a problem of size \( n \). Thus

\[
S_{\text{rel}} = \frac{T_{\text{par}}(n, 1)}{T_{\text{par}}(n, p)}
\]

Speedup means how much faster an algorithm runs using \( n \) processors compared to one. Usually we want to have an algorithm that executes twice as fast if we double the number of processor. If the speedup is below one it means that the algorithm actually executes slower than using one processor. If we divide the relative speedup with the number of processors we get efficiency, a measure of how good the algorithm utilizes the processors. If the efficiency is below one then not all processors have useful work to do, or the communication overhead is large. [16]

Absolute speedup is defined as the the ratio between the execution time of a (best known) sequential algorithm, \( T_{\text{seq}}(n) \), and the execution time of the parallel algorithm running on \( p \) processors, \( T_{\text{par}}(n, p) \). Which means

\[
S_{\text{abs}} = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n, p)}
\]
2.1.3 Parallelizing for shared memory machines using OpenMP

OpenMP [23] is an extension of both C and Fortran for writing parallel programs for shared memory machines. By writing special instructions you allow certain parts of your code to execute in parallel on different threads. To compile programs with OpenMP instructions, compiler support is needed; for instance, later versions of GCC and ICC support OpenMP. The following code piece has two function calls which will be executed in parallel.

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
        foo(x);
    }
    #pragma omp section
    {
        foo(y);
    }
}
```

In OpenMP the code will run in sequential until it enters a parallel region containing work-sharing constructs like do and sections; then new threads will be spawned and executed in parallel. In OpenMP there exist constructs for making variables private to threads and also work-sharing constructs like parallel for loops.

2.1.4 Parallelizing for distributed memory machines using MPI

Message Passing Interface [24], MPI, is a library for both C and Fortran for providing message passing between distributed memory machines. Instead of starting new threads on different CPUs/cores, as in the case with OpenMP, all processors or cores run the same program the whole time. Making them execute different functions is done by control statements like if, else etc. Each processor has its own identification, called rank. For creating a real world working MPI program, in principle only six MPI functions are needed [15]:

- **MPI_Init** for initializing the message passing interface.
- **MPI_Finalize** for terminating.
- **MPI_Comm_size** for getting the groups size (number of available processors).
- **MPI_Comm_rank** for getting the actual ID of the processor executing the call.
- **MPI_Send** for sending messages.
- **MPI_Recv** for receiving messages.
Of course there are many more MPI functions, for example to distribute data in an easy way to many processors, and also to divide processors into subgroups. Usually it is much more tricky and error prone work to write MPI programs than OpenMP programs, since programming of all communication between processors has to be done manually.

2.2 Component software

The idea of using components in software development is natural, at least if you see software development as an engineering discipline. Components have been used in different fields for a long time. In the late 1960s this approach was introduced for software engineering by a group of leading computer scientists at a meeting in Garmisch-Partenkirchen [5]. Intuitively, software components can be seen as building blocks of software that can be plugged together and form a larger software piece.

One of the first steps towards component software was made by McIlroy with his influence on the Unix operating system’s concept of pipes and filters. The idea is straight forward but anyhow powerful and flexible. By having standardised interfaces, virtually all programs (components) can be plugged together [5]. The idea is to have small programs designed for doing their task really well, and nothing more, so that the level of reuse becomes high.

We may ask our selves why we take up and discuss component software, when what we really want to do are transformations on source code. We will try to answer this question in the following sections.

2.2.1 Levels of composition

The pipes and filter approach, introduced by McIlroy, is called blackbox composition due to the fact that the components are never changed, and the user making the composition never has to bother what actually happens inside the component. Whitebox composition is the opposite, here you change the source code of a component when reusing it to fit your needs. The combination of whitebox and blackbox is greybox composition, where predefined places in the source code are made changeable but not everything. For transforming Quick Sort, whitebox or greybox composition could be possible ways.

2.2.2 Component systems

Today there exist component systems that are used in industry, for example CORBA and EJB which could be classified as blackbox systems. In the last years there has been some academic research done on the concepts of component systems, e.g. by Assmann and his colleagues. In [5] Assmann claims that component systems generally have three basic properties:

1. The component model: what kind of components can replace each other?

2. The composition technique: what operations can we use?
There are of course other ways to classify composition systems. One way is to distinguish if the composition is \textit{symmetric} or \textit{asymmetric}. In symmetric composition both composers and components are the same kind of first-class entities, but in asymmetric composition they are not \cite{1}. In practice this means that with a symmetric composition system it is possible to compose the composers by having a composer as an input to an other, not just connect them. For a deeper comparison between symmetric and asymmetric composition see Harrison et al. \cite{4}.

One further example of a specific component system, beyond the two ones we mentioned above, is ISC that we will describe in more detail in the next section.

\section*{2.3 Invasive Software Composition}

The basic idea behind Invasive Software Composition (ISC), compared to other software composition systems, is that ISC composes components by transforming them, not only put the components together. The transformation of the components makes them more reusable and is done invasively and no manual composition is needed \cite{5}. Assmann describes ISC \cite{5} by defining the following properties:

\textbf{Definition: Model of Invasive Software Composition.} A fragment box is a set of program elements. A fragment box has a composition interface that consists of a set of hooks. A hook is a point of variability of a fragment box, a set of fragments, or positions that are subject to change. A composer is a program transformer that transforms one or more hooks for a reuse context.

\textbf{Definition: Principle of Invasive Composition} In invasive software composition, composers instantiate, adapt, extend and connect fragment boxes by transforming their hooks.

\textbf{Definition: Implicit Hook.} An implicit hook is a set of program elements or positions that are contained in every component by definition of its programming language. All implicit hooks of a component make up its default composition interface.

\textbf{Definition: Declared Hook} A declared hook is a subset of program elements or positions in a fragment box that have been declared to be subject to change. All declared hooks of a component form its declared composition interface. \cite{5}

An example of implicit hooks could be the entry point of a function or the exit point whereas a declared hook adds extra information to a component used for composition. In \cite{5} it is stated that implicit hooks are not sufficient for composition. A declared hook can be seen as a parameter of a component, in the same way as functions have parameters. Usually function parameters have type, name and values, so do declared. The difference between function parameters and declared hooks is that function parameters contain runtime values, but declared hooks are
bound to code fragments. A fragment box can be seen as a special component that is used for invasive composition. [5]

ISC is state-of-the-art symmetric graybox composition. It is also the key to what we want to do, namely transform a program to a new program, since ISC can transform components. A normal source code fragment, in the ISC sense, is a component with only implicit hooks, yet it gives powerful transformation opportunities which we can use for our needs.

2.4 Reuseware

Reuseware [10] is a modelling framework that implements the concepts of ISC. Its goal is to "[..] provide composition technology and techniques to formal languages lacking such built-in mechanisms" [10]. Reuseware is developed as a plug-in for Eclipse and operates on the source code before it should be compiled. In this master thesis we will use Reusewares capability to describe languages with an EBNF-like grammar and its composition abilities. In practice Reuseware is used in two steps, in our case it will be the following two. [3].

The first step is to set up the composition environment by describing the language that our serial code is written in, in our case C, by giving its meta-model as a grammar with concrete and abstract syntax. We have to do the same for our parallelization language that we will use in the composition process, as well as for our extension (reuse language) of C, reuseC. From these grammars we can, by some simple mouse clicks, both generate parsers and language plug-ins that will be used in the next step.

The key to be able to transform a C program, in the next step, is the reuse language. The reuse language declares constructs, based on the C grammar, that can be parameterized. We have, for example, in our reuse grammar a rule that defines the syntax for a PrimaryExpression slot:

```
PrimaryExpressionSlot ::= "<|" "PrimaryExpression" name "|>";
```

This allows to write code like `<|PrimaryExpression _r|>` inside ordinary C code. For example we could create a statement like:

```
fragmentlist c.Statement ret_malloc=
'ret=(int*) malloc(sizeof(int)*(<|PrimaryExpression _r|>)+1);'.rc;
```

where _r should be replaced with a PrimaryExpression. This is not (yet) a valid C statement, but when connecting the PrimaryExpression _r with an existing expression it will be

```
bind _r on ret_malloc with

st_p1.expression.assignement.expression.expression.
expression.expression.expression.expression.expression.expression.
expression.expression.primaryexpression;
```
2.4 Reuseware

where \( st_p1 \) is a part of an existing abstract syntax tree. The \texttt{PrimaryExpression}
\_\texttt{r} could be seen as a declared hook and \texttt{fragmentlist c.Statement ret_malloc}
as a fragment box, if we refer to the notations of ISC. It is easy to see the similarities
with programming in a mainstream programming language. We create an object
of "type" \texttt{fragmentlist c.Statement} and give it a name and a value. The value
is a string of C code, in the above case a statement, that is extended with some
parameters where we later on can plug-in other objects.

The second step concerns the actual parallelization by using the generated plug-ins. Here we have to give
the actual serial code we want to parallelize, in our case study the Quick Sort program, a component. We also have to create
composers that are used for different parallelization strategies. Last we have to
provide is composition program that is written in our parallelization language and
that invokes the composers. The composition program is actually not needed, ev-
everything we want to do can be done without it. But writing composition programs
raises the level of abstraction, and could in some cases lead to higher reusability.
The language that the composers are written in is called \texttt{FraCoLa} [11], it is very
limited and it supports different ways of modifying programs at syntax tree level.
It can be used independently of the component language. If everything works as
planned we can then do a 'Build', the output will be the parallelized version of
our serial algorithm.

These two steps of how to use Reuseware are schematically described in Figure 2.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.1.png}
\caption{Overview of how Reuseware is invoked in our case.}
\end{figure}
2.5 Invasive Interactive Parallelization

"IIP is aimed as an automated mechanism that isolates the development of the core from the development of the parallelizing code" [1]. Chalabine states the following properties of his IIP model [1]:

\( P_1 \): The programmer receives automatically inferred assistance in selecting strategies.

\( P_2 \): Serial code is preserved in its original form, abstracted from the parallelizing statements.

\( P_3 \): Maintenance of the core and the parallelizing code are isolated and require different expertise.

\( P_4 \): Parallelization is turned into an interactive semi-automated on-demand process that is easy to activate and de-activate.

\( P_5 \): Parallelization becomes adaptable, i.e. supports an easy transition between parallelizing for shared- and distributed-memory platforms.

\( P_6 \): Parallelizing code is kept in a form where larger blocks can be reused.

These properties will be the definition of IIP that we will use. In practice an IIP system consists of three parts [1, Ch.2.6]:

1. Interaction engine
2. Parallelization engine
3. Weaving engine

These three parts will be realized by the Reuseware plug-in and the Eclipse framework. Reuseware lacks support of the first part, the interaction engine. We work around this problem by having variables that the user has to change inside the composers. Even if there is no interactivity the result will be exactly the same, and its just a matter of taste if the user is asked a question or has to 'check' inside the composers for variables to change. The variables that should be changed by the user are clearly marked inside the composers.

2.6 Divide and conquer

The idea of divide and conquer is to divide a problem into smaller, independent sub-problems that can more easily be solved, e.g. recursively, and then put together the results into a solution of the original, larger, problem. The algorithm is divided into three steps [8]:

- **Divide step:** The problem is divided into smaller sub-problems.
- **Conquer step:** Solve the sub-problems, either directly or recursively.
2.7 Quick Sort

- **Combine step:** Combine the results to a solution to the original problem.

It is natural to do divide and conquer recursively, doing a new divide step in the conquer step. This is the case for Quick Sort, see Section 2.7. It is also easy to see the potential of parallelism, since the sub-problems are independent and can be solved in parallel, although the implementation, in most cases, is far from trivial. See [8, 9] for a summary of parallel Quick Sort algorithms and implementation issues.

## 2.7 Quick Sort

Quick Sort was discovered by C. A. R. Hoare and first published in 1961 [12]. The following Quick Sort algorithm is taken from [15].

### Algorithm 1 Quick Sort

**Require:** Input Array $A$

**Require:** Input $r$ last element position, $q$ first element position

```plaintext
if $q < r$ then
    $x \leftarrow A[q]$
    $s \leftarrow q$
    for $i = q + 1$ to $r$ do
        if $A[i] \leq x$ then
            $s \leftarrow s + 1$
            swap($A[s], A[i]$)
            swap($A[q], A[s]$)
        $QuickSort(A, q, s - 1)$
    $QuickSort(A, s + 1, r)$
```

The algorithm can be described in three steps:

1. Choose an element (a pivot) from the array, the $x$ in the above code.

2. Reorder the array so all other elements smaller or equal than the pivot are to the left in the array, and all larger elements are to the right, and the pivot in between.

3. Recursively do it again, both on the part of the array that contains the larger elements and the smaller, until each sub-array contains one element.

Since the Quick Sort algorithm both yields independent subproblems that are smaller in size than the original problem and these subproblems are solved recursively, Quick Sort can be classified as a divide and conquer algorithm.

---

1 In [15] the Quick Sort algorithm actually contains an error that we corrected, see also http://www.imada.sdu.dk/~daniel/parallel/Errata_kirk.pdf.
Chapter 3

Implementation

The main work, concerning time consumption, is represented by this chapter. Our implementation was divided into two steps, first doing manual parallelization of Quick Sort for both shared and distributed memory machines, using OpenMP and MPI respectively, and the second step was to implement the composers used to parallelize Quick Sort. The last section of this chapter contains some personal reflections of using Reuseware. Since it is a tool under development, it has both features and bugs that we think have affected our work in a non-optimal way.

3.1 Manual parallelization

Before we can start to implement the actual transformation schemes used in Reuseware, we have to implement the parallel versions for QuickSort.

3.1.1 OpenMP

With OpenMP it all seems to be quite easy. At a first glance the intuitive implementation was just to parallelize the region containing the recursive calls, so they each would spawn a new thread and execute in parallel. This approach is also mentioned in [15] as well as its drawbacks, for example that the partitioning of the array is done in serial. When testing this approach for small arrays the speedup was limited. For larger arrays, around 5000 elements and more, it got a segmentation fault when testing on Solaris running on a dual processor, (UltraSPARC-IIIi) and compiling with GCC 4.2.0. A problem also occurred for the same problem size on Mozart, for technical specification see Section 4.1.1, also using two processors but compiling with ICC 9.1. Instead of reporting a segmentation fault Mozart reported that the program was aborted. Due to the drawbacks and these problems we choose another approach.

In an article by Suess et al. [14] the problems of parallelizing Quick Sort with OpenMP are described. The first problem they address is recursion. By using a stack they transform recursion into iteration, but also state
This step was one of the most time consuming of all, as it involves the introduction of a new data structure, specially tailored to our problem. For us this could be outside the scope. Suess also gives an alternative solution to the recursion problem which involves nested parallelism, but also states that nested parallelism is not supported by all OpenMP compilers and he also had to introduce a tracking mechanism for the number of running threads. The third solution he proposes uses a work-queue-model which is only supported by two compilers.

The implementation approach we finally choose is called ”Shared-Address-Space Parallel Formulation” and taken from [15]. Given an array of \( n \) elements, \( p \) processes and \( n/p \) elements per processor it could be summarized as follows:

1. Determine and broadcast the pivot.
2. Locally rearrange the assigned block into two sub-blocks, one for larger and one for smaller elements.
3. Determine the locations in the globally rearranged array that the local elements will go to.
4. Perform the global rearrangement.
5. Recursively do these steps again, for the smaller and the larger elements.
6. Recursion ends when a sub-block is assigned only one processor and done by sequential sort.

Our implementation uses two help functions, \( \text{LocalRearrange} \) and \( \text{GlobalRearrange} \). \( \text{LocalRearrange} \) is called by each processor, with its specific parameters. Each processor has its own part of the whole array. It locally rearranges the elements so that these smaller or equal than the pivot are stored in beginning of the sub-array, and the larger ones in the rest of the sub-array. It also counts the number of smaller elements and larger ones and stores them in two variables, respectively. \( \text{GlobalRearrange} \) puts all the locally partitioned elements into a new large array of size \( n \), so all larger elements are to the right and smaller ones to the left. This could be done in parallel. For simplicity the prefix sums for the displacements are calculated in inside \( \text{GlobalRearrange} \), which means each processor calculates its own. Of course it could been done with parallel prefix sums, but it would have introduced synchronization problems. After running \( \text{GlobalRearrange} \) the new array has to be copied to the original array of size \( n \). Now all elements, smaller elements to the left and larger to the right, are in the original array. For an example of these steps see Figure 3.1. Depending on how many elements were smaller and larger than the pivot we can now decide how many processors will be used for running \( \text{GlobalRearrange} \) the parallel Quick Sort again on the part of the smaller respectively larger part of the array. This is done in such way that, for example, if 40% of the elements are smaller than the pivot, 40% of the processors will be assigned the smaller elements, the rest of the processors will be assigned the rest of the elements. Actually it is not optimal to proportionally assign the processors to the
sub-arrays depending on the number of elements. The proper way would have been to do it in relation to how much work the processors have to perform for each sub-array. The problematic issue is that the number of processors that can be assigned to the sub-arrays are integer numbers, so even if the work can be estimated it is still a problem to map the processors in good way. Solutions to this have been proposed by Eriksson et al. based on serialization and repivoting [9].

![Initial array:](image)

**Figure 3.1.** Example of first iteration when running LocalRearrange and GlobalRearrange using five processors on an array with 20 elements.

This algorithm could be classified as a SPMD (single program, multiple data) algorithm since all processors are running the same program on different data and the number of threads remains fixed throughout execution (no dynamic spawning of threads or processors). The Local- and Globalrearrange functions are run on all available processors. Actually the algorithm is a group SPMD algorithm because for each recursive step the new sub-arrays are assigned to smaller and smaller processor groups, until the each group only contain one processor.

### 3.1.2 MPI

Our implementation of Quick Sort for distributed memory is mainly based on an already existing implementation in C++ made by M. Chalabine, which had to be ported to plain C. Even though we had an existing implementation to start with it needed a lot of debugging. We are not sure if we would have got our version working on a large set of cores without using the debugging environment Totalview [19].

The algorithm is as follows:
To begin with all processors get \( n/p \) (rounded) elements. All processors are in the same group (MPI_COMM_WORLD). Group size is the number of processors in the group. Each processor in a group has a unique ID called rank (an integer).

1. Call Quick Sort for the group:
   
   (a) If group size > 1
      
      i. Processor 0 in the group selects a pivot and sends it to the other processors in the group.
      
      ii. All processors in the group partition their data in such way that the elements smaller than and equal the pivot go to the left in the array and larger or equal elements go to the right.
      
      iii. All processors in the group calculate the split point, which means the number of processors that are assigned to the smaller elements. Split point = Group size * (number of smaller elements/(total number of elements assigned to the group))\(^1\).
      
      iv. All processors in the group calculate how many processors they will receive from; processors getting smaller elements than pivot get elements from processors to the right of the split point, and vice versa.
      
      v. Define and create new groups left and right, where processors with rank < split point go to the left and larger or equal rank to the right group.
      
      vi. Distribute data: First each processor has to compute how many elements it is going to receive from each other one and also calculate the displacement in the receive buffer. Then the processor sends the small elements to processors with rank smaller than split point and vice versa.
      
      vii. Call Quick Sort recursively; if rank < split point, call it for the left group otherwise call it for the right group. This will be executed in parallel.

   (b) If group size <= 1, this means only one processor is assigned to the sub-array.
      
      i. Do sequential sorting. Each processor stores how many elements it has got.

2. Inform processor 0 how many elements it should receive from each processor by letting each processor send this information.

3. Processor 0 gathers the array from the other processors.

### 3.2 The composers

In the implementation of the composers we have chosen two slightly different approaches. The main reason for this is to show the wide range of capabilities that

\(^1\)This is not an optimal split point, see discussion in Section 3.1.1
the invasive part of IIP has. Since we have two cases of parallelism, one for shared memory machines and one for distributed memory, we apply different approaches to each case. In the shared memory version we do one monolithic transformation: one piece of input code will be transformed to an output code. Here we study just one function, namely the QuickSort itself, and produce a parallel version. In this case we analyze the code and transform it. Here we do both changing of original code, as well as adding extra functions, by including code from an extra file that we already have written. Including information from a file or keep it inside the composer is just a matter of taste of where to store the information. If we add code fragments from a file or if this information is stored inside our composer itself, makes no difference, one or the other way is can be more practical depending of what you want to do. In the second case we go up to the main function level, and do not only analyze the QuickSort function itself. This gives us extra opportunities for using different composers, that together will give a complete functional transformation. Another reason for including parallelization statements on a higher level in the call structure is that MPI initialization should be done in the main function. Some of the composers are more general than others, and could possibly be used in different contexts, which also gives reuseability.

3.2.1 OpenMP

We have chosen to make one monolithic basic composition program that transforms the serial Quick Sort to a parallel OpenMP version. Our main motivation for this approach is that we walk through the syntax tree and search for the control structures where we want to do different operations in the same place. First of all we have to recognize that a given code is actually Quick Sort. We will not depend on function or variable names; instead we will analyze the control and data dependence structure. We have, for simplicity, only focused on our particular implementation and its control structure concerning if-statements, for-loops and recursion. The structure is

```c
if expression
{
    variable declarations
    for-loop
    {
        if expression
            if-body
    }
    recursive call
    recursive call
}
```

Transforming Quick Sort to Localrearrange

When doing a parallelization transformation we add a lot of information, this extra information has to be stored somewhere. In our approach we will use the original
Quick Sort and then transform it to LocalRearrange, described in Section 3.1.1, since they have a lot in common, then we have to provide the rest of the code and append it. If we compare the following LocalRearrange function to Quick Sort, Section 2.7, it has almost the same structure, but no recursive calls are done and some extra parameters are used for counting the numbers of smaller and larger elements.

```c
void LocalRearrange(int *A, int q, int r, int pivot, int *S, int *L )
{
    int x, s, i;
    (*S)=0;
    (*L)=0;

    if(q < r)
    {
        x= pivot;
        s= q;

        if(A[q]<=x)
            (*S)++;

        for(i=q +1; i<=r; i++)
        {
            if(A[i]<=x)
            {
                s++;
                (*S)++; // to keep count of how many smaller or eq. elements
                myswap(&A[s],&A[i]);
            }
        }
        myswap(&A[q],&A[s]);
    }
    (*L)=r-q+1-(*S); //number of larger than pivot elements.
}
```

First of all we load the whole file, and assume the first function in the file is Quick Sort. Normally this could be guided by the users in an interactive session. Then we pick out the function parameters, we assume three of them, and store them in variables. We add the extra function parameters that we need; *S, *L and pivot. S is for keeping count of number of smaller elements and L for larger. Pivot has to be an input parameter because all processors should partition around the same pivot. So we have replace the old pivot, $x$ in Quick Sort in Section 2.7 and inject the statements for incrementing $S$ and $L$. We also detect and remove recursion by picking out the identifier name of the function and look for function calls to the same name. To be able to know which of the parameters in the function are the array, left and right element we can pick out left and right in the if condition,
since we have a less-than condition, then the last parameter is the array. The last things we do is to inject the helper functions to file. This composer has 195 lines of code, for implementation see Appendix A.1.

Problems

Now we try to match a certain set of criteria (a pattern) that we think describes the Quick Sort algorithm. Problems can occur when matching something that is actually not Quick Sort. Our solution to this is IIP, before the actual transformation occurs we should ask the user if this is Quick Sort. A more general approach would be to ask if the matched function is a divide and conquer algorithm. When using the IIP approach the user is always part of the matching process. Some ideas for matching a general divide and conquer algorithm are discussed in Section 4.2.

3.2.2 MPI

Here we chose to divide the problem into sub-problems, even though the transformation of the algorithm was done very aggressively. After recognizing the Quick Sort algorithm by matching the control structure in the same way as in the OpenMP case we insert a parallel version. This is the largest transformation step. Still we do the transformation in several steps, to add modularity that could be reused. These steps could be described as a pipes and filter approach, where the output of one composer will become the input to the next. The final output depends on the order of how the filters are run, and applying them in wrong order could produce non-functioning code.

We assume that all functions, both Quick Sort and main, are in the same file.

Implementation with a parallelization language

In Reuseware it is possible to write a so called basic composition. It is a program written in FraCoLa. To load a code piece that can be transformed, you explicitly write the file name: \texttt{fragmentlist c.TranslationUnit help =/source.c;} in the composition code.

To get a higher level of modularity you instead write composers that instead take the code piece as a parameter and the output is also in some sense a parameter. The code itself is written in FraCoLa. To invoke those parameterized composers a language is needed, in our case we call it \texttt{parallelization language}. To make a composition a small program in the parallelization language is written that calls the composer. We also added instructions to our parallelization language to support each composer. For us the parallelization language does not give any extra power, only an extra level of abstraction and overhead. Still this way is preferred in general to make composition more user friendly.

In the following sections each composer is described. The two components mpi-init and mpi-finalize add basic constructs and calls:
mpi-transform

We use the same approach for recognizing Quick Sort as in the OpenMP case. Instead of changing the code we insert a parallel version of QuickSort that we already have. The first step is to load the file, which is a parameter to the composer, containing both Quick Sort and the main function. We assume that Quick Sort is the first function and the main function the last, even though it is possible to search for the main function. We assume that Quick Sort has three parameters and we store them in variables, then we do as in the OpenMP case; map them to the array, right element and left element. One difference is that we both have to care about the parameters of the Quick Sort function used in the declaration, as well as the variables used in the call of Quick Sort function, done from the main function. This means that we have to care about six variables.

The next thing we have to do is to create a return array, to be appended in the main function, for storing the sorted elements. It is dynamically allocated and the size is calculated using the variables used for the Quick Sort call. Now we can insert both statements for the allocation of the array and the call to the parallel Quick Sort, and delete the old call. The last step is to append the help functions and the parallel version of Quick Sort.

This composer has 276 lines of code, see Appendix A.2.

mpi-init

MPI initialization should be applied as early as possible in the function hierarchy, which means in the beginning of the main function. Before the actual MPI_Init call can be done we have to include `#include <mpi.h>` in the source file. It should be included before the main function and Quick Sort function, we add it in the beginning of the source file. The next step is to introduce a new variable, `rank`, for holding information about each processor. The next step is to inject `MPI_Init(&argc, &argv)`; where `argv` and `argc` are the standard parameters of the main function. After the MPI initialization call is injected we have to create a call for setting the rank: `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

We have to do a remark concerning introducing a new variable, in this case `rank`. This variable will be used by some of the other composers, and will therefore become a dependency between them. The other composers that use rank as a special variable have to "know" in some way that it is this special name. This can be solved in different ways, for example the following.

1. Assumption, we assume that rank as variable name is used and its special. This can create name clashes with other, already existing, variables and is therefore not a preferred solution.

2. Introduce it in special a way, maybe introduce a set of special variables that are used for interaction between composers, for example `myMPI_rank`.

3. Using namespace or define statements, this is more or less an extension of 2.
4. Ask the user about "special" variables. This means user interaction, and fits well to the IIP approach. The question could, for example, be "Which variable is used to carry information about the rank?"

5. Factor out definition of rank into a separate composer with well-defined (standard) name.

This composer has 38 lines of code, see Appendix A.3.

**mpi-finalize**

This composer takes a C source file as input and as output the same program but with a MPI_Finalize() statement in the end of main. MPI_Finalize() should be as high up in the program hierarchy as possible, that is the reason why we add it in the main function. It assumes that MPI initialization already is done. All MPI function calls should be dynamically between MPI_Init() and MPI_Finalize(). Adding MPI_Finalize(); in the end of a program is easily done with Reuseware, we just look up the last statement and inject the statement after it. Analysing the problem a bit further we realize that MPI_Finalize() should be executed before program termination. The main function could have multiple exit points; a function normally terminates after a return-statement, or when the whole block has been executed. In our case we look if there is a return statement in the end of the main function or not. For simplicity in our implementation we do not do any more checks if there are return statements anywhere else. We assume that the program always reaches the end of the main function before terminating. Programs can of course also terminate in a wrong way, for example after an unhandled exception. We do not bother at all in these cases of doing MPI_Finalize() in a correct way. However, a thorough transformation should handle such cases appropriately, in principle, or prompt the user where it cannot recognize all exits of the main function properly. This composer has 37 lines of code, see Appendix A.4.

**mpi-print-with-rank**

As a result of the programming style of MPI the same program is executed on all processors. Making different processors execute different code pieces (in the same program) is achieved by using control mechanisms, for example depending on their rank (id) they do either send or receive. Sometimes programs are cluttered with printf-statements, and when converting from serial code to parallel code you have to take care of the printouts. Normally this is done by only letting one processor do printing, since that execution order in most cases can not be determined beforehand. This composer that we introduce makes a wrapper around all printf-statements, controlling that only the processor with rank 0 makes the call. The rank variable is introduced with the mpi-init composer. What this composer actually finds is a function call with the name "printf" and it doesn’t care about the parameters. Instead of hardcoding which processor, in our case 0, should print, asking the user in an IIP manner which processor should do the printouts could be a further development. This approach could also be used in a more general
context in some cases, for instance, to encapsulate any function calls that should only be done by one processor. This composer has 37 lines of code, see Appendix A.5.

**mpi-init-array**

This composer initializes an array with random numbers on the processor with rank 0. This could be seen more or less as a helper function. The size, name and location where to inject the initialization code piece is hard coded, this could theoretically be solved by IIP where the user will be asked about size, name and where to inject the code. This composer has 35 lines of code, see Appendix A.6.

**mpi-print-array**

This composer adds code for printing an array. The size, name and where to inject it is hardcoded but could be solved by IIP. This combined with the mpi-print-with-rank composer gives that the array will be printed only on processors with rank 0. These two composers are a simple example of how composers can be used together to achieve a desired result. This composer has 27 lines of code, see Appendix A.7.

**General discussion**

What we could have done is having one composer that recognizes Quick Sort, and then reuse it for both MPI and OpenMP versions. The main reason against this solution is that compositions of composers are very limited in Reuseware, according to an email conversation with the Reuseware developers. Instead we use the pipes and filters approach, and when recognizing Quick Sort we do at the same time some transformation steps. These transformations are sometimes different for MPI and OpenMP versions.

### 3.2.3 Implementing in Reuseware –personal reflections

The learning curve of Reuseware is quite hard, the main reason is the lack of documentation. The existing documentation is in my opinion well written but there should have been more of it; the example dealing with EBNF grammars is small. This leads to a lot of trial and error development. One factor that makes it hard to learn as well is that the grammar is to be provided by the user; one nice feature would have been that some standard grammars for example for C and Java are provided together with Reuseware. There are several reasons for this, first of all dealing with and writing grammars is hard and takes time. Secondly, how the grammar is written will influence how hard it will be to use it in the composition step. Even if the grammar is correct and works, there maybe exist better implementations that will make it easier to use in the composer. One example is the following.

```java
if(mystatement0.expression.assignement.expression.expression.expression.
expression.expression.expression.expression.expression.expression.expression.
tmp instanceof c.FunctionCall)
```
where we check if a certain statement is a function call. This is of course error
prone and in practice when working with it you must have your grammar in front
of you.

One other thing is that the programming language, FraCoLa[11] used for pro-
grramming the composers, is very limited. The "data types", if we can call them
so, are the grammatical types. In the language no "regular" types exist, not even
boolean, which could be nice for storing data used inside the composers.

Maybe we misunderstood some things, but at the step when the parallelizing
language is defined it is also defined how the composers will be invoked using the
parallelizing language. The composers themselves are implemented in the next
step, using another instance of Eclipse and Reuseware. This is not logical in our
view; first we need to decide how to invoke something that is not yet written. It is
also really unpractical to have to go back and change the parallelization language
each time a new composer is created to be able to use it.

In the normal case we run two instances of Eclipse, one for modelling the
language, the other one for making the compositions. Since Reuseware is imple-
mented in Java it uses the Java virtual machine. Both Reuseware and Eclipse are
memory hungry, when running a composer a lot of RAM is used, more than 1 GB
is not uncommon.

The implementation of Reuseware is buggy when running language generators.
Reuseware often reports errors. Usually these are solved by deleting ALL the
old generated models, both in the project view and in the file system, and then
regenerate them from scratch. And they will work just fine!

Sometimes Reuseware complains on your input programs (components) that
you know worked yesterday! Deleting all generated language models and generate
them again, will usually solve the problem. Also restarting Reuseware before
regenerating the language models will also help!

One problem that really affects the usability is that the code that comes out
from the composers has a lot of line breaks, for example

for (
    i = q
    + 1
    ;
    i
    <= r
    ;
    i
    ++
)

which of course is annoying for a human to read, even if it is valid C code. In our
OpenMP version this caused problems, since OpenMP is an add-on to C and not
plain C. In OpenMP actually the line breaks are of importance. The following code piece

```c
#pragma omp for
for (i = 0; i < p; i++)
```

is a piece of output from one of our composers. This will compile until the `for` statements are on different lines, which has to be solved manually after running the composers. Another problem this creates is that it makes it hard to compare different implementations in terms of lines of code. The parallel, transformed, OpenMP version of Quick Sort is due to this 762 lines of code and the MPI version is 2665. In [2] preservation of textual structure of the input program when applying ISC weaving transformations is described, which could be a solution to the above problem.

All of the above problems have during the implementation process been more or less frustrating and we think that they in some cases have slowed done the implementation process considerably.
Chapter 4

Results and discussion

This chapter will focus on the result in terms of speedup as well as stating that our hypothesis holds. We also compare our solution to related work and finally we address future work.

4.1 Results

Here we will go through both how good our parallel versions of Quick Sort work concerning speedup and if we fulfill the requirements stated in Chapter 1.

4.1.1 Speedup

Our main goal has not been to implement state of the art, highly tuned parallel versions of the Quick Sort algorithm. Instead we wanted to test if IIP could provide the means of (semi) automatically parallelizing Quick Sort. Anyhow we will in this section provide some figures that show that reasonable speedup can be achieved by our parallelization approach.

It took around 2 minutes to run the longest transformation, which was the transformation chain for parallelization for MPI, invoking the six composer using a moderate laptop (IBM Thinkpad R50e, Pentium M 735 1.7 GHz and 1 GB RAM). This is in our opinion a bit too slow, at least if it should be used interactively.

All tests for the MPI version were done on Neolith [20], which is a Linux cluster consisting of 805 HP ProLiant DL140 G3 servers. Each server has two quad-core processors of type Intel® Xeon® E5345. The total peak performance is 60 Tflops. The MPI environment we used was OpenMPI [22].

All the tests for the OpenMP version of Quick Sort were done on Mozart [21], which is a SGI Altix 3700 Bx2, having 64 Intel Itanium 2, 1.6 GHz processors and 512 GB shared main memory.

We measured for both the MPI and OpenMP implementations the speedup using different number of processors and problem sizes, $n$, 1, 10 and 100 million elements. Both implementations gain some relative speedup, even though the
results are not so good. For a better MPI implementation, which is using load balancing, see [8].

In the shared memory case running our parallel algorithm gives the same execution time as running a qsort fully in sequential. For the distributed memory version there exists a difference, see Table 4.1.

### Table 4.1. Fully sequential version of C qsort compared to MPI version running on one processor

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Exec. time (s) fully sequential</th>
<th>Exec. time (s) parallel on one processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10^6$</td>
<td>0.237</td>
<td>0.312</td>
</tr>
<tr>
<td>$n = 10^7$</td>
<td>2.4966</td>
<td>3.7766</td>
</tr>
<tr>
<td>$n = 10^8$</td>
<td>27.9984</td>
<td>42.3802</td>
</tr>
</tbody>
</table>

Another issue is that the speedup figures also are depending on problem size. Compare Tables 4.6 and 4.7. This shows the general problem of implementing good parallel algorithms that works satisfactory for different cases of granularity.

### Table 4.2. Parallel Quick Sort OpenMP speedup, $n = 10^6$, using C qsort in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4784</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.2708</td>
<td>1.766</td>
</tr>
<tr>
<td>4</td>
<td>0.3686</td>
<td>1.298</td>
</tr>
<tr>
<td>6</td>
<td>0.3658</td>
<td>1.308</td>
</tr>
<tr>
<td>8</td>
<td>0.3766</td>
<td>1.270</td>
</tr>
</tbody>
</table>

### Table 4.3. Parallel Quick Sort OpenMP speedup, $n = 10^7$, using C qsort in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.451</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5.046</td>
<td>1.278</td>
</tr>
<tr>
<td>4</td>
<td>3.602</td>
<td>1.791</td>
</tr>
<tr>
<td>6</td>
<td>3.973</td>
<td>1.624</td>
</tr>
<tr>
<td>8</td>
<td>4.778</td>
<td>1.350</td>
</tr>
</tbody>
</table>

### 4.1.2 Correctness

In order to convince ourselves that our parallel implementations of Quick Sort actually work we had to check that the sorted arrays were correct. It was also a crucial step during the manual parallelization. Doing test runs on small arrays
### Table 4.4. Parallel Quick Sort OpenMP speedup, $n = 10^8$, using C `qsort` in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76.102</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64.800</td>
<td>1.174</td>
</tr>
<tr>
<td>4</td>
<td>52.077</td>
<td>1.461</td>
</tr>
<tr>
<td>6</td>
<td>73.020</td>
<td>1.042</td>
</tr>
<tr>
<td>8</td>
<td>63.915</td>
<td>1.191</td>
</tr>
</tbody>
</table>

### Table 4.5. Parallel Quick Sort MPI speedup figures, $n = 10^6$, using C `qsort` in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.312234</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0.136065</td>
<td>2.29</td>
</tr>
<tr>
<td>16</td>
<td>0.308471</td>
<td>1.01</td>
</tr>
<tr>
<td>32</td>
<td>0.637559</td>
<td>0.49</td>
</tr>
</tbody>
</table>

### Table 4.6. Parallel Quick Sort MPI speedup figures, $n = 10^7$, using C `qsort` in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7766</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1.15549</td>
<td>3.27</td>
</tr>
<tr>
<td>16</td>
<td>1.1295</td>
<td>3.34</td>
</tr>
<tr>
<td>32</td>
<td>1.0082</td>
<td>3.75</td>
</tr>
<tr>
<td>64</td>
<td>2.1152</td>
<td>0.66</td>
</tr>
</tbody>
</table>

### Table 4.7. Parallel Quick Sort MPI speedup figures, $n = 10^8$, using C `qsort` in the sequential base case.

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec. time (s)</th>
<th>Rel. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42.3802</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>31.3556</td>
<td>1.35</td>
</tr>
<tr>
<td>3</td>
<td>18.6075</td>
<td>2.28</td>
</tr>
<tr>
<td>4</td>
<td>18.6821</td>
<td>2.27</td>
</tr>
<tr>
<td>5</td>
<td>15.5047</td>
<td>2.73</td>
</tr>
<tr>
<td>6</td>
<td>13.9095</td>
<td>3.05</td>
</tr>
<tr>
<td>7</td>
<td>13.3264</td>
<td>3.18</td>
</tr>
<tr>
<td>8</td>
<td>12.0776</td>
<td>3.51</td>
</tr>
<tr>
<td>16</td>
<td>9.05515</td>
<td>4.68</td>
</tr>
<tr>
<td>32</td>
<td>6.20810</td>
<td>6.83</td>
</tr>
</tbody>
</table>
and printing out the whole sorted array was natural. On large arrays we checked that the arrays actually was sorted by comparing elements using a loop.

### 4.1.3 Fulfilling the requirements

In Chapter 1.1.3 we stated four requirements that could be summarized as: our composer(s) should transform Quick Sort to parallel versions for both shared and distributed memory using pattern matching. We think we have fulfilled these requirements.

### 4.1.4 Fulfilling the hypothesis

In our hypothesis we stated that using ISC and IIP we can parallelize divide and conquer problems and at the same time get acceptable speedup for both shared and distributed memory machines.

In the shared memory version with problem size $n = 10^8$ we get a speedup of 6.83 using 32 processors, this is comparable to [9]. They get a speedup of 6.63, when no load balancing is done, using the same problem size and same number of processors. Even though we measure relative and they absolute speedup, we think the figures are comparable. In the distributed memory version the speedup is lower. In [14] execution times for different implementations for OpenMP are listed. In their implementation that uses nested parallelism they archive a speedup\(^1\) of 1.12 and 1.93 using two processors and four processors respectively when sorting $10^8$ elements. We think these figures are comparable to our.

In Chapter 2 we gave a brief overview of the background needed for doing this. In Chapter 3 we did two actual implementations, one for shared memory and one for distributed memory, using ISC and IIP, for a given divide and conquer problem, namely the Quick Sort algorithm. In Section 4.1.1, we also gave some speedup figures for the two implementations, that we consider acceptable. Our conclusion is that the speedup figures and correctness check confirms that our hypothesis holds.

Since we have showed that it is possible parallelize an instance of the Quick Sort algorithm we think that it would be possible to write composers that can parallelize other implementations of Quick Sort and other instances of divide and conquer algorithms. In Section 4.2 some methods are described, and their drawbacks.

The main reason of why our approach could be generalized to other divide and conquer algorithms is the power of IIP. IIP can fill the gap between automatic and manual parallelization and therefore cover more instances of divide and conquer. This case study is one step towards parallelizing divide and conquer algorithms in general.

\(^1\)Calculation of speedup from wall-clock time is done by us.
4.2 Related work

Today there exists a lot of work that contributes to how to solve the parallelization problem. In Section 2.1.2 we gave examples of methods of how parallelization usually is done. Here we will discuss some more specific concepts and compare them to IIP.

Using aspects

In [1] Chalabine compares the work by Harbulot on how to use aspects for parallelizing loops with IIP. One big difference is that Harbulot focuses only on Java, while the IIP approach is more general since the language is an input. IIP does not focus on one construct, while Harbulot focuses on loops. IIP can be used to combine several concerns into one composition [1]. Another issue concerning aspects is that they are asymmetric while IIP and ISC in general can handle both symmetric and asymmetric composition. [1]. In the case of Reuseware, to be honest, the composition is not fully symmetric since a composer itself can not be input to another composer.

Manual parallelization

If we compare IIP to manual parallelization there are in our opinion two main advantages with IIP, it is simple and a lot of work is automatized. On the other hand, since a parallelization expert must have wide knowledge about hardware architecture, algorithms and the actual problem, it is in most cases hard for an automatic parallelization tool to compete when it comes to an overall view of the whole problem and its solutions.

Automatic parallelization

Fully automatic parallelization is done by the compiler, its main advantage is that the programmer does not have to be aware of parallelization at all, and the parallelization process is fully hidden. This is good, especially if the programmer is a domain expert and has limited knowledge about parallel programming. The drawback is usually that the compiler only exploits a subset of the available parallelism since it can only do conservative assumptions and static analysis of the code [1]. For some examples of automatic parallelization see page 33. If we compare automatic parallelization to IIP, the advantage is that IIP lets the user provide knowledge about the program, so the parallelization can be done more aggressively and exploit more of the available parallelism. We think that IIP takes the best from both manual and automatic parallelization.

Skeletons

Skeletons are black box components with well defined interfaces that are parameterizable. In [1] two main problems are mentioned:
Manual change of the code is needed, a serial program has to be rewritten to use skeletons.

Skeletons only work satisfiably if the program is well structured.

Another issue with skeletons is that, even if each component is well tuned and works in an optimal way, this is not always the case if two or more components are combined to solve a larger problem. Since IIP is not black box composition it allows more changing of the components than skeletons do. Therefore it is more likely that composition of several parallel components will in total give a better code when using the IIP approach. In our opinion IIP maybe easier to use than skeletons since no manual refactoring is needed using IIP and is supported by a graphical tool. It could also be possible to use IIP to restructure a program into skeleton form.

Matching and parallelizing divide and conquer algorithms

To be able to match a divide and conquer algorithm in general, it must be clear how the algorithm will look like. As stated in Section 2.6 a divide and conquer algorithm has three steps and will have the following structure:

• The problem size has to be an input (implicit or explicit).
• Check if the problem is small enough to be solved as a base case.
• Divide the problem into smaller and independent sub-problems.
• Recursive function calls for the sub-problems.
• Combine the sub-solutions.

A divide and conquer algorithm can in general be written as follows, where \( n \) is the problem size.

\[
f(..., n) \\
{ 
  if(n < C) 
    ProcessBaseCase() // Base case 
  else{ 
    Divide(n, n1, n2,..., nd) // Divide original problem into sub-problems 
    f(..., n1) // Solve the sub-problems 
    . 
    . 
    . 
    f(..., nd) 
    Combine(); // Merge the sub-solutions 
  } 
}
\]
4.2 Related work

The problem size \( n \) can of course be implicit and \( n1, n2, ..., nd \) should each be smaller than \( n \). It is also hard to be sure that the \( d \) recursive calls are independent. In some cases the base case is implicit and also the combine part of the algorithm, as in the Quick Sort case in Section 2.7. There is nothing that says that no other if-statements can occur before or after the test if the problem size is smaller than a certain value (threshold). To be able to transform a general divide and conquer algorithm we have to focus on the three phases and independently transform them. Compared to our implementation this is a totally different problem. In our case we instead identified an instance of a divide and conquer algorithm and applied some transformation rules that are specific to the instance.

However the problem of parallelization of divide and conquer algorithms has been addressed before, in [26] Mattson et al. explain it as a pattern with a solution. In an article by Rugina and Rinard [27] they give a method of automatically parallelizing divide and conquer algorithms that uses dynamically allocated memory. Their approach is to analyze the pointers in the algorithm in several ways. They analyze the bounds of the memory accesses for function calls to be able to conclude if the function calls are independent. Then the code that their compiler generates gives the possibility for the runtime system to execute the independent calls in parallel. In their implementation they use the SUIF compiler infrastructure and parallelize the code using the Cilk-statements spawn and sync. When they parallelize a certain sorting algorithm they get a speedup of 4.4 using eight processors. Compared to our parallelization approach they are more general since they do not try to match a given algorithm. Another issue is that they do not do use the same aggressive transformation approach as we do. Their approach shows how hard it is to match divide and conquer algorithms, since what they do is not to match a whole divide and conquer algorithm, instead they give a way to find out which of the function calls are independent and can be executed in parallel. Of course this approach works well with divide and conquer, due to the fact that the calls are independent by definition. Their solution works only with shared memory machines; maybe it is possible to expand it to distributed memory even if it would require injection of more complicated statements. Anyway it would be interesting to test if their approach could be repeated using Reuseware.

A great work concerning pattern-driven parallelization has been done by Kessler [18] that resulted in PARAMAT, an automatic parallelization tool. PARAMAT has 150 patterns for algorithms like linear algebra, numerical integration and Simplex. Many scientific programs usually use a limited number of operations. PARAMAT exploits this but also that the instructions usually are data parallel and can be described with patterns. Combinations of patterns can then form new patterns. The pattern matching is done on the abstract syntax trees of a program in a bottom-up way. After the pattern matching is done optimized parallel code is generated, including local algorithm replacement. PARAMAT concerns scientific code, which in this case is operations performed inside loops. PARAMAT does not include a divide and conquer pattern, and does not match recursive calls. Anyhow it is much more powerful than Reuseware when it comes to pattern matching, and its approach could maybe be extended for implementing a pattern for parallelizing divide and conquer algorithms as well.
4.3 Future work

To make Reuseware a useful composition system in everyday work we think that our parallelization language has to be defined and developed in a structured way, which includes keywords and how to name and invoke different types of composers etc. In our opinion this work has to be done carefully since designing a language requires a lot of knowledge about how the language will be used. When it comes to implementation we think that making a library of patterns that recognize different versions and implementations of the Quick Sort or similar algorithms will be of great help; today we only recognize one that matches one control structure. Of course patterns for other sorting algorithms could be put into the library as well. A lot more work can be put into tuning the parallel Quick Sort algorithms, also here a library with different implementations depending user provided information could be useful. One alternative approach of using IIP is that there shouldn’t be a one to one mapping between serial and parallel code. For example if a composer recognizes mergesort, it could be transformed to the best known sorting algorithm in the library depending on what architecture specification the user provides. Resources could for example be number of cores or threads that are available. This could be taken even further as suggested by Kessler and Löwe in [17], where the actual algorithms can be switched during runtime depending on problem size and available cores etc. Even the schedule can be varied depending on context information as introduced by Eriksson et al. in [9].

Another interesting area is to make composers for SIMDization of code; some work has been done concerning SIMDization for the CELL processor using skeletons by Ålind et al. [28].
Bibliography


[26] Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill, Patterns for Parallel Programming, Addison-Wesley Professional, USA, 2005

Appendix A

Source Code

Here some of the sources will be presented that are of special interest, namely the composers used for the transformations.

A.1 qs_transform2.bc

This is the source code for the composer used for transforming Quick Sort to parallel Quick Sort for OpenMP.

```c
#define composer pl.QSParallelizerOMPtest(core) {
    // it still keeps the old function name..
    //eriha4458@student.liu.se
    fragmentlist c.TranslationUnit myfunc = core;
    fragmentlist c.TranslationUnit help =/helpfunc.c;

    fragmentlist c.Identifier funcN = myfunc.declaration[0].declarator.directdeclarator.identifier;

    //pick out the parameters, Assume 3 of them
    fragmentlist c.Identifier first_p = myfunc.declaration[0].declarator.directdeclarator.
    tmp.parametertypelist.parameterlist.declaration.declarator.directdeclarator.
    identifier;
    fragmentlist c.Identifier second_p = myfunc.declaration[0].declarator.directdeclarator.
    tmp.parametertypelist.parameterlist.more[0].declarator.directdeclarator.identifier;
    fragmentlist c.Identifier third_p = myfunc.declaration[0].declarator.directdeclarator.
    tmp.parametertypelist.parameterlist.more[1].declarator.directdeclarator.identifier;

    //Add parameters needed in the function header: pivot, *S, *L
    fragmentlist c.ParameterDeclarationDirect mypivot = 'int pivot'.rc;
    fragmentlist c.ParameterDeclarationDirect myS = 'int *S'.rc;
    fragmentlist c.ParameterDeclarationDirect myL = 'int *L'.rc;
    append myfunc.declaration[0].declarator.directdeclarator.tmp.parametertypelist.
    parameterlist.more[last] with mypivot;
    append myfunc.declaration[0].declarator.directdeclarator.tmp.parametertypelist.
    parameterlist.more[last] with myS;
    append myfunc.declaration[0].declarator.directdeclarator.tmp.parametertypelist.
    parameterlist.more[last] with myL;

    //Detects & removes recursion. Initialize *S & *L
    foreach(mystatement : myfunc.declaration.compoundstatement.statements.statement) {
        //First level in the function
        if (mystatement instanceof c.IfStatement) {
            //now we are inside if statement if(q<r) {...}
            //Here we actually check if its a less-than-condition
            if (mystatement.condition.assignment.expression.expression.expression.
                expression.expression.expression.more.operator.instanceof c.Less) {
                //Now we can assign right and left & figure-out the array!
                //This is the way to go:
            }
        }
    }
}
```
fragmentlist c. Identifier Left = mystatement.condition.
  assignment. expression. expression. expression. expression. expression.
  expression. expression. expression. expression. expression.
  expression. expression. primaryexpression;
fragmentlist c. Identifier Right = mystatement.condition.
  assignment. expression. expression. expression. expression. expression.
  expression. expression. expression. expression. expression.
  expression. expression. expression. primaryexpression;

fragmentlist c. Statement Lassign = '('+L)=('<|PrimaryExpression
  _right|>−('<|PrimaryExpression _left|>+1−(*S)');'.rc;
bind _right on Lassign with Right;
bind _left on Lassign with Left;
append myfunc.declaration.compoundstatement.statements.
  statement[ last] with Lassign;

//Now we can decide which of the parameter that is the array
if (((Left equals second_p) && (Right equals third_p)) ||(( Left
  equals third_p) && (Right equals second_p))){
  fragmentlist c. Identifier Array = first_p;
} else {
  if (((Left equals first_p) && (Right equals second_p))
      ||(( Left equals second_p) && ( Right equals first_p )))
  {
    fragmentlist c. Identifier Array = third_p;
  } else {
    if (((Left equals first_p) && (Right equals third_p))
        ||(( Left equals third_p) && ( Right equals first_p )))
    {
      fragmentlist c. Identifier Array = third_p;
    } }
}
// Initiate *S & *L to zero...
fragmentlist c. Statement setSzero = '(*S)=0;'.c;
fragmentlist c. Statement setLzero = '(*L)=0;'.c;
prepend mystatement with setSzero;
prepend mystatement with setLzero;

//figure out the old-pivot
foreach (mystatement2 : mystatement.body.statements.statement)
{
  if (mystatement2 instanceof c.ForLoop)
  {
    foreach (mystatement3: mystatement2.body.statements.statement)
    {
      if (mystatement3 instanceof c.IfStatement)
      {
        //This part is for increasing the S counter...
        fragmentlist c. Statement splusplus = '(*S)++;'.rc;
        append mystatement3.body.statements.statement[ last] with splusplus;
      }
      //pick out the the old pivot...
}
if (mystatement2 instanceof c.ExpressionStatement)
{
  if (mystatement2.expression.assignement instanceof c.LogicalOrExpression)
  {
    if (mystatement2.expression.assignement.expression.expression.expression.expression.expression.expression.
        expression.expression(expression.expression.expression).
tmp instanceof c.FunctionCall)
    {
      fragmentlist c.PostfixExpression myfkncall = mystatement2.expression.
        assignement.expression.expression.expression.expression.expression.
        expression.expression.expression.expression.expression.expression;

      if (funcN equals myfkncall.primaryexpression)
      {
        //Delete recursion (this will be done twice)
        extract mystatement2;
      }
    }
  }
  
  if (mystatement2 instanceof c.ExpressionStatement)
  {
    if (mystatement2.expression.assignement instanceof c.RealAssignmentExpression)
    {
      if (mystatement2.expression.assignement.left.primaryexpression.equals old_pivot)
      {
        //now we should delete the old and append the new.
        //x=A[q] => x=pivot ;
        fragmentlist c.Statement new_pivot_assign = '<|'.rc;
        bind _i1 on new_pivot_assign with old_pivot;
        bind _i2 on new_pivot_assign with pivot;
        append mystatement4 with new_pivot_assign;

        //We should also add a control structure
        fragmentlist c.IfStatement controlls = 'if (<| PrimaryExpression _array|>|[<| PrimaryExpression
          _element|>]<| PrimaryExpression _oldpivot|>) {('.
        bind _array on controlls with Array;
        bind _element on controlls with Left;
        bind _oldpivot on controlls with old_pivot;

        //creates trouble if we extracted...
        prepend mystatement4 with controlls;
        extract mystatement4;
      }
    }
  }
}
//Not perfect with parameter txy, but it works...
fragmentlist c. FunctionDefinition LRA = 'void LocalRearange (int txy) {<|Statement _ss |} \'} rc;

//fragmentlist c. DirectDeclarator Par = myfunc.declaration [0]. declarator.
directdeclarator.tmp. parametertypelist. parameterlist.declaration. declarator. directdeclarator;
fragmentlist c. Statement functionBody = myfunc.declaration [0]. compoundstatement;
//bind _xy on LRA with Par;
bind _ss on LRA with functionBody;
append LRA.declarator.directdeclarator.tmp.parametertypelist.parameterlist.more[last] with myfunc.declaration [0]. declarator.directdeclarator.tmp.parametertypelist.parameterlist.declaration;
foreach (parameter : myfunc.declaration [0]. declarator.directdeclarator.tmp.parametertypelist.parameterlist) {
    //print LRA.declarator.directdeclarator.tmp.parametertypelist.parameterlist.more to /hej4.c;
    append LRA.declarator.directdeclarator.tmp.parametertypelist.parameterlist.more[last] with parameter;
}
//extract LRA.declarator.directdeclarator.tmp.parametertypelist.parameterlist.
declaration.declarator.directdeclarator.tmp.parametertypelist.parameterlist.declaration;
append help.declaration [7] with LRA;
extract myfunc.declaration [0];
foreach (decl : help.declaration) {
    append myfunc.declaration[last] with decl;
}

A.2 mpi-transform.bc

This source code transforms Quick Sort partly to a parallel MPI implementation. For full functionality other composers have to be applied as well, see Section 3.2.2.

define composer pl.MPITransform (core){
    //Load the inputfiles:
    //fragmentlist c. TranslationUnit myfunc = /orgQS_2.c;
    fragmentlist c. TranslationUnit myfunc = 'core';
    //fragmentlist c. TranslationUnit myfunc2 = /mpi_final3.c;
    fragmentlist c. TranslationUnit myfunc2 = /mpi_final3V08.c;
    //Assume that QS is the first function declaration
    fragmentlist c. Identifier funcN = myfunc.declaration[0]. declarator.directdeclarator.
declaration;
    //Assume that main is the last function (it is possible to search for it as well with MPIInit.bc)
    fragmentlist c. Identifier mainf = myfunc.declaration[last]. declarator.directdeclarator.
declaration;

    //For picking out the variables used in the call to QS
    fragmentlist c. Statement st_p1 = '<|PrimaryExpression _pe| >; '.rc;
    fragmentlist c. Statement st_p2 = '<|PrimaryExpression _pe| >; '.rc;
    fragmentlist c. Statement st_p3 = '<|PrimaryExpression _pe| >; '.rc;

    //Is main always the last function?
    //This could easily be searched for...
    foreach (mystatement0 : myfunc.declaration[last]. compoundstatement.statements.statement) {
        if (mystatement0 instanceof c. ExpressionStatement) {
            if (mystatement0.expression.assignment instanceof c. LogicalOrExpression) {
                if (mystatement0.expression.assignment.expression.assignment instanceof c. FunctionCall) {
                    fragmentlist c. PostfixExpression myfkncall0 = mystatement0.expression.assignment.expression.expression.expression.expression.expression.expression.expression.expression.assignment.fn instanceof c. FunctionCall;
                }
            }
        }
    }
}
Here we detect the quicksort call in the main-function. If we should save it for use later on:

```c
if (funcN equals myfkncall0)
  primaryexpression
{
  fragmentlist c. PrimaryExpression
    call_p1 = myfkncall0.
    tmp. parameter . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . primaryexpression;
  fragmentlist c. PrimaryExpression
    call_p2 = myfkncall0.
    tmp. parameter .more[0]. expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . primaryexpression;
  fragmentlist c. PrimaryExpression
    call_p3 = myfkncall0.
    tmp. parameter .more[1]. expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . primaryexpression;

  bind _pe on st_p1 with call_p1;
  bind _pe on st_p2 with call_p2;
  bind _pe on st_p3 with call_p3;
}
```

// Stupid workaround
```c
fragmentlist c. StatementList RecList = '<| Statement something|>'.rc;
bind _something on RecList with mystat;
extract RecList. statement [ first ];
```

// Stupid workaround again!
```c
fragmentlist c. StatementList myList = '<| Statement something|>'.rc;
bind _something on myList with mystat;
extract myList. statement [ first ];
```

// Pick out the function parameters, Assume 3 of them
```c
fragmentlist c. Identifier first_p = myfunc. declaration[0]. declarator . directdeclarator.
  tmp. parametertypelist . parameterlist . declaration . declarator . directdeclarator .
  identifier;
fragmentlist c. Identifier second_p = myfunc. declaration[0]. declarator . directdeclarator.
  tmp. parametertypelist . parameterlist .more[0]. declarator . directdeclarator . identifier;
fragmentlist c. Identifier third_p = myfunc. declaration[0]. declarator . directdeclarator.
  tmp. parametertypelist . parameterlist .more[1]. declarator . directdeclarator . identifier;
```

// Pick out the variables used in the call to quicksort
```c
//fragmentlist c. Identifier call_p1
//fragmentlist c. Identifier call_p2
//fragmentlist c. Identifier call_p3
```

// We want to check that the function actually is Quicksort
```c
// Try to match:
// If control
  // for loop
  // if control
```
// recursive call
// recursive call
// As well as figure out, which of the variables
// that are the array, left, and right
// in the first if control

foreach (mystatement : myfunc.declaration.compoundstatement.statements.statement)
{
    if (mystatement instanceof c.IfStatement)
    {
        // Here we actually check if it's a less-than-condition
        if (mystatement.condition.assignment.expression.expression.expression.expression.expression.expression.expression.expression.expression.expression.expression.expression.expression.assignment.operator instanceof c.Less)
        {
            // Now we can assign right and left & figure out the array!
            // This picks out the right ones, but with "wrong" type
            // This is the way to go:
            bind _xyz on q_less_r with mystatement;
        }
    }

    foreach (mystatement2 : mystatement.body.statements.statement)
    {
        if (mystatement2 instanceof c.ForLoop)
        {
            foreach (mystatement3 : mystatement2.body.statements.statement)
            {
                if (mystatement3 instanceof c.IfStatement)
                {
                    append myList[last].statement with mystatement3;
                }
            }
        }
        if (mystatement2 instanceof c.ExpressionStatement)
        {
            if (mystatement2.expression.assignment instanceof c.LogicalOrExpression)
            {
                if (funcN equals myfkncall.primaryexpression)
                {
                    append RecList.statement[last] with mystatement2;
                }
            }
        }
    }
}
New call to the parallel mpi quicksort, replaces old call...

```
fragmentlist c.Statement pqs_call = ret = pqs(rank, <|PrimaryExpression_elements|>, <|PrimaryExpression_r|> - <|PrimaryExpression_q|> + 1); rc;
fragmentlist c.Statement ret_malloc = ret = (int *) malloc(sizeof(int) * ( <|PrimaryExpression_r|> - <|PrimaryExpression_q|> + 1 )); rc;
```

```c
}}
}

else
{
  if (third_p equals Right)
  {
    bind _r on pqs_call with st_p3.expression.assignment.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression;
    bind _r on ret_malloc with st_p3.expression.assignment.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression;
  }
  }

//Now we can decide which of the parameter that is the array
//We also use this to figure out which of the parameters in the quicksort call
// are the array and so on..
if (((Left equals second_p) && (Right equals third_p)) ||
((Left equals third_p) && (Right equals second_p)))
{
  fragmentlist c.Identifier Array = first_p;
  bind _elements on pqs_call with st_p1.expression.
    assignment.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.expression.
    expression.expression.expression.
    primaryexpression;
}
else
{
  if (((Left equals first_p) && (Right equals second_p))
    || ((Left equals second_p) && (Right equals first_p)))
  {
    fragmentlist c.Identifier Array = second_p;
    bind _elements on pqs_call with st_p2.expression.
      assignment.expression.expression.expression.
      expression.expression.expression.expression.
      expression.expression.expression.expression.
      expression.expression.expression.expression.
      expression.expression.expression.expression.
      expression.expression.expression.expression.
      primaryexpression;
  }
  else
  {
    if (((Left equals first_p) && (Right equals third_p))
      || ((Left equals third_p) && (Right equals first_p)))
    {
      fragmentlist c.Identifier Array = third_p;
      bind _elements on pqs_call with st_p3.expression.
        assignment.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        primaryexpression;
    }
  }
}

//if the new call was created correctly we can extract the old and append the new:
if ((isbound _elements on pqs_call) && (isbound _r on pqs_call) && (isbound _q on
pqs_call))
{
//Create a return int array
fragmentlist c.Declaration incl_ret = 'int *ret'; c;
append myfunc.declaration[last].compoundstatement.declarations.declaration[last] with
incl_ret;

foreach (mystatement0 : myfunc.declaration[last].compoundstatement.statements.statement)
{
  if (mystatement0 instanceof c.ExpressionStatement)
  {
    if (mystatement0.expression.assignment instanceof c.LogicalOrExpression)
    {
      if (mystatement0.expression.assignment.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression.expression.expression.expression.
        expression,tmp instanceof c.FunctionCall)
      {
        fragmentlist c.PostfixExpression myfncall0 =
          mystatement0.expression.assignment.
            expression.expression.expression.
            expression.expression.expression.
            expression.expression.expression;
        }
A.3 mpi-init.bc

This is the source code for the composer used for appending MPI_Init(). For discussion see 3.2.2.

define composer pl. MPIInit( core ) {
    //Here we add MPI_Init stuff
    //We always work on the main file, since
    //MPI initialization should be applied as
    //early as possible in the function-hierarchy
    fragmentlist c.TranslationUnit myfunc = core;
    fragmentlist c.Identifier mymain = 'main'. rc;
    fragmentlist c.PrPrIncludeGlobal includeMPI = '#include <mpi.h>'. c;
    if (not myfunc.declaration contains includeMPI) {
        prepend myfunc.declaration[first] with includeMPI;
    }
    foreach( functiondeclaration : myfunc.declaration ) {
        //Seem to evaluate from left to right in the conditions, this means
        //that the order is IMPORTANT and leads to "COMPLICATED" logic. Think twice!
        if ( functiondeclaration instanceof c.FunctionDefinition &&
            functiondeclaration.declarator.directdeclarator instanceof c.
            DirectDeclaratorIdent &&
            functiondeclaration.declarator.directdeclarator.identifier equals
            mymain ) {
            fragmentlist c.Declaration incl_rank = 'int rank;'. c;
            fragmentlist c.Statement mpi_init = 'MPI_Init(&argc, &argv);'. c
            fragmentlist c.Statement mpi_rank = 'MPI_Comm_rank( MPI_COMM_WORLD, &rank);'. c;
            append functiondeclaration.compoundstatement.declarations.declaration[last] with incl_rank;
            prepend functiondeclaration.compoundstatement.statements.statement[first] with mpi_init;
        }
    }
    //print result, remember we also need to run MPIFinalize.bc on the outputfile
    //to make it complete
    //print myfunc to /outOK.c;
}
A.4 mpi-finalize.bc

This is the source code for the composer used for appending `MPI_Finalize()`. For discussion see 3.2.2.

```c
#define composer pl . MPIFinalize(core) {
  // Adds `MPI_Finalize()` statement in the end of a main file
  // nothing fancy
  fragmentlist c. TranslationUnit myfunc = core;
  fragmentlist c. Identifier mymain = 'main'. rc;
  foreach(functiondeclaration : myfunc. declaration)
  {
    // Seem to evaluate from left to right in the conditions, this means
    // that the order is IMPORTANT and leads to COMPLICATED logic. Think twice!
    if (functiondeclaration instanceof c. FunctionDefinition &&
        functiondeclaration . declarator . directdeclarator instanceof c. DirectDeclaratorIdent
        && functiondeclaration . declarator . directdeclarator . identifier equals mymain)
    {
      fragmentlist c. Statement mpi_finalize = 'MPI_Finalize();'. c;
      // return statements can be anywhere, this don’t find them all of them
      // we assume that the program terminates at the end of the main-function
      // MPI_Finalize should be as high up in the program hierarchy as possible.
      if (functiondeclaration . compoundstatement . statements . statement[last] instanceof c. ReturnStatement)
      {
        // is there a return statement in the end of the main function??
        // Yes:
        prepend functiondeclaration . compoundstatement . statements . statement[last] with mpi_finalize;
      }
      else
      {
        append functiondeclaration . compoundstatement . statements . statement[last] with mpi_finalize;
      }
      //print mpi_finalize to /out2.c;
    }
  }
}
```

A.5 mpi-print-with-rank.bc

This is the source code for the composer used for only letting one processor do printing. For discussion see 3.2.2.

```c
#define composer pl . PrintWithRank(core) {
  // Usually printouts are made by rank 0
  // we assume this is the case
  fragmentlist c. TranslationUnit myfunc = core;
  traverse (fragment : myfunc . declaration[last] . compoundstatement . statements . statement)
  {
    //if(fragment instanceof c. PostfixExpression )
    if (fragment instanceof c. ExpressionStatement)
    {
      if (fragment . expression . assignement instanceof c. LogicalOrExpression)
      {
        if (fragment . expression . assignement . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . expression . tmp instanceof c. FunctionCall)
        {
          fragmentlist c. PrimaryExpression funcN = 'printf '. rc;
        }
      }
    }
  }
```
expression . expression . expression . expression . expression . expression . expression . expression . expression ;
if (funcN equals myfkncall . primaryexpression)
{
fragmentlist c . IfStatement rank_add = ' if (rank==0)
fragmentlist c . IfStatement _ss = '<|Statement _ss|>'].rc ;
bind _ss on rank_add with fragment ;
prepend fragment with rank_add ;
extrait fragment ;
}
// print myfunc to /mpi7.c;
}
}

A.6 mpi-init-array.bc

This is the source code for the composer used initializing an array with random numbers using one processor. For discussion see 3.2.2.

define composer pl . InitArray (core) {

 // Initialize an Array with random number
 // Only processor with rank Zero does the initializing
 // Interactive approach:
 // user has to give name of array, size
 // and where to inject it.
 // Please set array_name & array_size;
 fragmentlist c . PrimaryExpression array_name = 'array '.c ;
 fragmentlist c . PrimaryExpression array_size = '1000'.c ;

 fragmentlist c . Statement init_func = ' if (rank==0){int i ; srand(time(NULL)) ; for (i=0; i<
<|PrimaryExpression _pr0|>; i++){ <|PrimaryExpression _pr|>[i ] = rand() %100;} }'.rc ;
bind _pr0 on init_func with array_size ;
bind _pr on init_func with array_name ;
 // myfunc.declaration[last] be injected.
 fragmentlist c . TranslationUnit myfunc=−
 append myfunc.declaration[last].compoundstatement.statements.statement[6] with
 init_func ;

 // print myfunc to /mpi5.c;
}

A.7 mpi-print-array.bc

This is the source code for the composer used initializing an array with random numbers using one processor. For discussion see 3.2.2.

define composer pl . PrintArray (core) {

 // PrintArray
 // This adds statements for printing an array
 // Only processor with rank Zero does the printing;
 // Interactive approach:
 // user has to give name of array, size
 // and where to inject it.
 // Please set array_name & array_size;

 fragmentlist c . PrimaryExpression array_name = 'ret '.c ;
 fragmentlist c . PrimaryExpression array_size = '1000'.c ;
 fragmentlist c . Declaration loop_counter='int r lc ; '.c ;

 fragmentlist c . Statement print_func = ' for (r lc=0; r lc< <|PrimaryExpression _pr0|>; r lc++){ printf("%i ", <|PrimaryExpression _pr|>[r lc]) ;} '.rc ;
bind _pr0 on print_func with array_size ;
bind _pr on print_func with array_name;
// Set the file and function as well as where the print statement
// myfunc.declaration[last] be injected
fragmentlist c.TranslationUnit myfunc=->core;
append myfunc.declaration[last].compoundstatement.declarations.declaration[last] with
loop_counter;
append myfunc.declaration[last].compoundstatement.statements.statement[6] with
  print_func;

// print myfunc to /mpi4.c:
}
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Erik Hansson