Proceedings of the 1994 International Workshop on Parallel Processing
Lessach, Austria, Sep 26–30 (ISSN 0943-3805)

Compilation and Efficient Parallel Execution of Recursive Programs

Stefan U. Hänßgen*
haenssgen@ira.uka.de

October 1994

Abstract

Many scientific and computational problems are naturally expressed as recursively branching programs. They exhibit a dynamically changing behavior both in computation and communication, which poses a challenge for parallel programming systems. Among those problems are for instance n-body calculations, divide-and-conquer algorithms and tree based programs.

This paper presents the design of a compiler and a runtime system for efficiently executing such programs in parallel. The user writes a recursive imperative program that the compiler analyzes and optimizes for parallel execution. Since many important parameters cannot be determined statically, the runtime system reports profiling data to the compiler that uses it for the next compilation.

By this combination of static and dynamic parameter gathering, the compiler can access more information than in conventional approaches, making further optimizations possible.

1 Introduction

Parallel programming is still no easy task, neither for the programmer nor for providers of programming tools. In many areas we see advancements – machine independent explicitly data parallel programming, for example, has been proven to be feasible and efficient for a wide range of problems [Phi93]. However, nearly all of these approaches run into severe trouble when recursive parallelism is used where the resulting dynamic behavior of the program, both in data alignment/distribution and load balancing, ruins performance. Compilers that perform automatic parallelization encounter even more severe problems.

This would not be a severe restriction if recursive parallel programs were only interesting for toy problems, but there are many real-life applications that are naturally expressed in a recursive way and offer parallelism at the level of a recursion step, leading to a tree-like branching of subproblems[1]. Examples of such applications are programs with recursive dynamic data structures, pure divide-and-conquer algorithms, and n-body problems that occur in molecular dynamics and astrophysical simulations and can be solved in $O(n \log n)$ time per step using recursive structures.

In this paper, we present the design of a system that can handle on parallel machines this kind of recursive programs that are both useful for real-life applications and difficult for existing systems. By deriving problem specific parameters, we plan to gain enough additional information to make an efficient parallel execution of these programs possible. The programmer only formulates the problem as a recursively branching program; the compiler extracts the parallelism

*This work was supported by the Graduiertenkolleg "Controllability of Complex Systems" (DFG Vo 287/5-2)
[1] We ignore linear recursion since its nature is sequential.

and transforms the program for efficient parallel execution. We do not need explicit parallel constructs since the parallelism can be derived from the recursive structure, so we gain complete machine independence at the language level. Our approach relies on bidirectional information exchange between the compiler and the runtime system to gather static and dynamic data on the program and thus enable several optimizations both at compile time and run time.

The rest of this paper is structured as follows: We give an overview of the whole system and describe the functionality of compiler and runtime system in section 2. There we also explain the different heuristics the system uses to improve parallel performance, show implementation issues, and describe how we plan to validate the system. In section 3, we give some examples of programs we want to handle and explain how the system can extract the necessary information from them. The related work is discussed in section 4, and finally, in section 5 we draw some conclusions and show future directions of research.

2 System Overview

The structure of the compiler and the runtime system is shown in figure 1 below. Tools and program transformation steps are drawn on the left, the information and methods used by the components are on the right. Note the parameter feedback loop from the runtime system to the compiler.

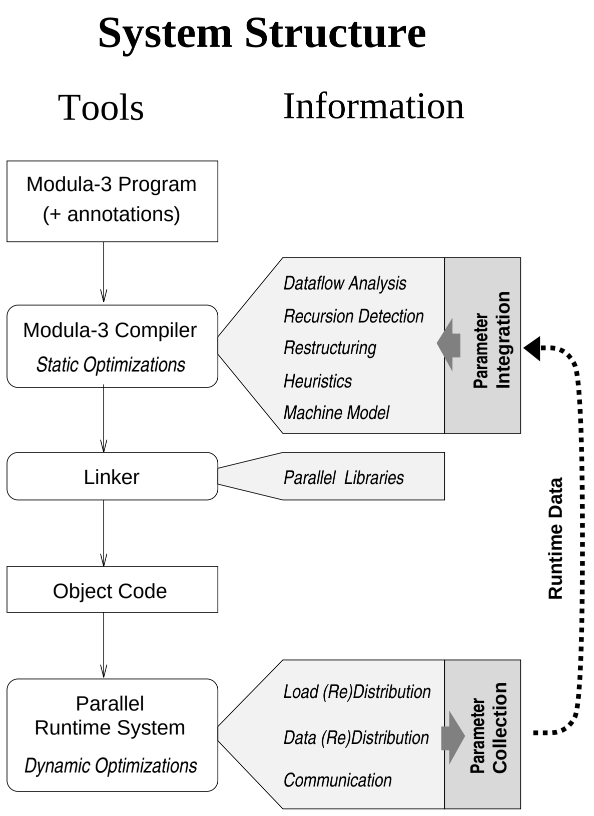

Figure 1: Compiler, runtime system, and their interaction.

In the following text, the term information means any kind of knowledge that the system can use to predict, estimate, or improve program performance. Concrete examples of such information and their use are given in section 2.3.

The underlying execution model is a parallel machine consisting of several interconnected processors that are capable of executing several program threads at once.

2.1 Compiler

The compiler reads the source code and tries to extract as much information as possible. For this, it uses standard methods such as data flow analysis and identification of data structure such as trees [Hen90]. Of course, in general there are many aspects of a recursive program that cannot be derived statically, e.g., the computation time of one recursion step or the recursion depth. To compute those parameters, the compiler instruments the program with code to obtain their values at runtime. The resulting data is then used during the next compilation run. In section 2.3 we will explain the parameters and their use in detail.

It might also be necessary to use program annotations for cases that cannot be efficiently measured or derived, for instance, stating obscure data (in)dependencies can help the compiler for further analysis. We do not further regard this issue in detail.

Using the information it has gathered, the compiler performs various optimizations such as loop unrolling, message prefetch, and communication bundling. It also supplies hints to the runtime system to facilitate dynamic optimizations.

2.2 Runtime System

To execute the program in parallel, we use a runtime system that takes care of issues such as resource allocation, process and data migration, and load balancing. To optimize those operations, it uses hints from the compiler derived from analysis or previous runs.

It also incorporates a machine model that describes communication (latency, bandwidth), power of one processing element, available memory, caches, and thread costs (creation, switching). These factors result in a cost function for each program that we want to minimize. For example, on a machine with fast communication it is not so expensive to distribute computations, while on a machine with powerful processors but low bandwidth we want to keep everything local. The specific weights for the machine model used are obtained once for each parallel machine in a special profiling run.

Apart from the functionality mentioned above, the runtime system also dynamically measures the parameters about which the compiler requested information and saves them for the next compilation run.

2.3 Optimization Heuristics

In this section we take a closer look at the program parameters mentioned above. They are specifically selected for recursive programs on parallel machines. Based on their values, the compiler and the runtime system can answer strategic questions such as "should we create another execution thread on this processor or move some work to a remote processor?". We claim that this additional knowledge allows for more efficient parallel execution.

Of course, the more we know about a parameter, the more exact our estimations and predictions of the program's respective behavior will be, and the better we will be able to optimize it. We distinguish five different kinds of parameter values, depending on how much we could find out:

Constant – the value never changes, e.g., a program always generates 4 subproblems in a recursion step (or ends the recursion, of course).

Runtime constant – the exact value is not known at compile time but remains fixed once it is set in the program, e.g., the recursion depth of a program may depend on the size of the input data. A sophisticated compiler could even derive a closed formula modeling this interdependence.

Limited – we do not know the value in advance, but we know upper or lower bounds, e.g., a program spawns up to 7 subproblems in each step but never more.

Distribution – we only have some statistical distribution of the parameter's value, e.g., the average recursion degree is 3.67 and the deviation is 0.52.

Completely unknown – we know nothing at all about the value.

In many cases, the compiler can derive the parameter's value statically, e.g., when it is evident from textual analysis of the program that the recursive call occurs twice or never. The compiler may also be able to estimate bounds. For all other cases, we use feedback from the runtime system and perform statistical analysis for parameter estimation[2].

In the following sections, we describe in detail each parameter, how we can obtain its value, and how the system uses this knowledge. We use small schematic pictures to illustrate the meaning of a parameter in the context of some tree-like computation structure where the nodes execute the recursive functions and the edges correspond to recursive calls.

2.3.1 Recursion depth

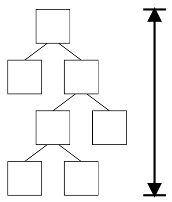

Figure 2: The recursion depth is the depth of the tree of recursive calls.

Detection: Statically, we can find out the recursion depth of a program by a heuristical pattern matching for a (runtime) constant start value and unconditional decrements, e.g., a function is called with a constant parameter i, in each recursive call i is decremented, and recursion stops depending on the value of i. Dynamically, we analyze the recursion depths of the last program run from the profile data.

Use: When we know the recursion depth it is easier to decide PE mapping and allocation both in advance ("how should I distribute the initial processes?") and at runtime ("there is just one more recursion step left, so we should keep it on this processor and save the cost for spawning another execution thread").

[2]This approach is based on the assumption that we can indeed derive parameters that are statistically significant, i.e., the program's behavior is not completely random or utterly dependent on the input data.

2.3.2 Recursion degree

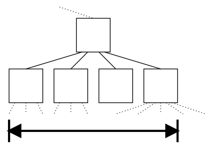

Figure 3: The recursion degree is the number of subproblems (children) generated in a recursion step.

Detection: Statically, we can detect loops with fixed boundaries that spawn off new computation or simply an unconditional sequence of recursive calls. Dynamically, we have to rely on empirical observations from runtime data.

Use: Knowing the recursion degree allows for better allocation strategies. We also know the (maximum) number of processes at a given tree depth, so we can enumerate them in advance and, given a processor allocation strategy, compute which processor owns which subtree without communication.

2.3.3 Computation per node

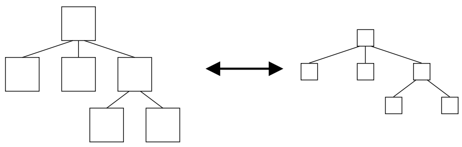

Figure 4: The computation per node is the amount of computation that one step of the recursion takes, ignoring communication costs.

Detection: Statically we can estimate the amount of computation when there are no or only fixed-length loops and function calls. Otherwise, we have to measure the computation at runtime with user-time stamps at the beginning and the end of each program section of interest. Thus, we ignore the system time that is used up in communication delays.

Use: We can estimate the program's granularity and make decisions like "can this processor handle one more computation or will it be overloaded?", i.e., trading distribution and communication costs against computation costs. This is useful both at compile time and at runtime, e.g., a program that does next to no computation at all, like summing a tree of numbers, should do most of its computation on one single node to avoid the costly communication. On the other hand, if one recursion step involves solving large systems of equations, we do not want many of those on one node. The weights for the corresponding machine model cost functions depend on the computer used, see above.

2.3.4 Original Data location

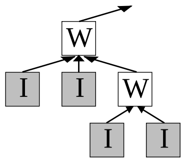

Figure 5: The data location tells us where in the computation tree the input and working data is located.

Detection:

If input data is only accessed at the bottom of a recursion, it is located in the leafs only. Working data generated from leaf data can be found in the nodes

Use: Knowledge about the data location allows for better data distribution and -alignment decisions and also communication prediction.

2.3.5 Communication paths

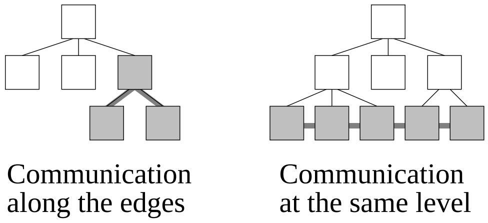

Figure 6: Communication paths describe which nodes exchange data.

The source program itself exhibits of course no explicit communication, but each data access to a non-local element means a communication operation at runtime that we have to take into account. Depending on the machine used, a good communication behavior is essential for performance.

Detection: We know that communication just takes place along edges of the computation tree if only data from the direct parent or child is used. Communication on the same depth level of a tree can be detected by expression analysis, depending on the data distribution. Furthermore, communication patterns can be recorded at runtime and analyzed empirically later.

Use: In general, we want to enhance locality and thus reduce communication. If only the edges of a tree are used in communication, we can be sure that all data stays local to a subtree, so we can migrate it at once. We can also try to group "families" of nodes that communicate much among themselves. Finally, such knowledge allows us to perform a better initialization and the application of techniques such as communication bundling and message prefetches.

2.3.6 Memory requirements of one step

The memory requirements of a recursion step are the spatial analogon to the computation per node parameter.

Detection: If there are constant allocations we know the memory requirements statically. Otherwise, we have to generate profile data dynamically.

Use: Like with the computation size, we can estimate the spatial granularity of a program and control allocation and distribution. Memory limits are more severe than computation power limits since they are absolute – if there is no more memory left, we cannot do much, while we can always add one more time slice to computation. Memory usage also hints to the minimum computation per node.

2.4 Implementation

To be able to test our ideas, we have to implement a system that uses them. The main area of research is not language design, although we may have to introduce some annotations as mentioned before, but optimization heuristics that are based on a close cooperation between compiler and runtime system.

As base language we chose Modula-3 since it is a widely used imperative language with a freely available modular compiler that is not overly difficult to modify. Also, researchers at our institute have already used and enhanced Modula-3[Hei93], so we can base our work on existing experience.

At first, we will concentrate on thread level parallelism on multiprocessor MIMD computers, possibly with shared memory. A portable runtime system will be based on top of Posix threads that are conveniently similar to Modula-3 threads. We do not plan to make use of any special interconnection topology such as grids. These decisions are based on the simplicity of programming for us and should not limit the generality of our approach.

2.5 Validation

Once the system produces working programs, it is necessary to validate our claim, namely that by information feedback from the runtime system and using special heuristics, we can improve the parallel execution of recursive programs.

To that end, we use a *benchmark* suite with real-life programs and perform measurements with and without optimizations enabled. This shows which optimizations result in which gains for which kinds of problems. Also, we test the *scalability* of our approach by using different numbers of processors and varying problem sizes.

These methods only produce results relative to the performance of our own system. For absolute comparison, we chose two ways:

Hand-optimized explicitly parallel programs are written and their performance is compared to the compiler generated programs. Of course, this assumes that "hand-optimized" is a fair comparison standard, i.e., we still take Modula-3 for programming but use our full knowledge of the problem to gain performance. The potential problems with this approach are that we can never be sure we have explored all possible optimizations and that it is not clear at which level to stop optimizing, e.g., algorithmic changes may speed everything up enormously but are no fair comparison to a compiler.

Competing systems are used for comparison, i.e., we rewrite programs of the benchmark suite for other parallel systems like Vienna FORTRAN and compare their performance with ours. This approach also has problems at least of practical nature – we have to get access to machines where both our system and the competing system run, or we have to scale the performance data. Also, most competing systems are not designed for recursive parallelism, so we have to try to work around that.

Taken together, these experiments will show how well our systems performs and if our claim is valid.

3 Examples

Now that we have explained how the system works, we take a closer look at some sample programs and show what the system can recognize and optimize and how it is done.

3.1 Tree Sum

The first example is the recursive summing of a tree of numbers. It is interesting because we encounter a recursive data structure. Also, only little time is spent in actual computation, so we have a good test for our machine model – should we sum it all on one processor and not communicate at all, or does it pay off to distribute data to several processors? The answer to this question also depends on the size of the input data which is a runtime constant that can be determined by compiler analysis.

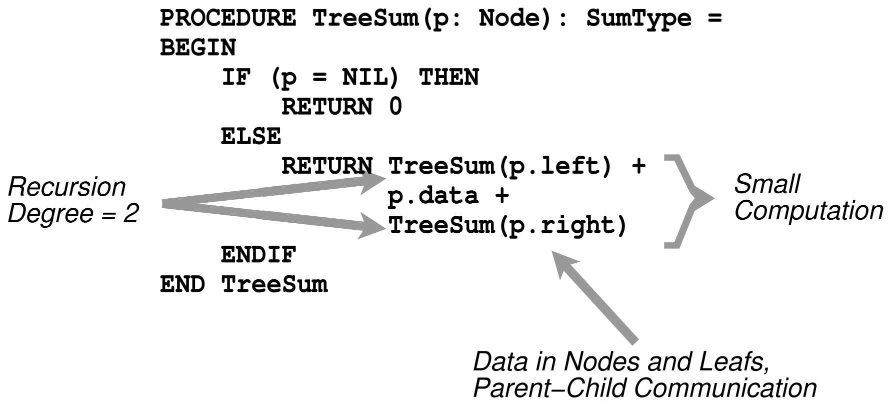

Figure 7: A compiler's view of the tree sum program.

The compiler can statically find out many of the parameters:

- The *recursion degree* is a constant 2, because in each step, either recursion is ended or two subproblems are generated.
- The *computation* itself is very small, just a single addition.
- *Data* is located initially in the leaves, and the sum itself is built up in the nodes.
- We have *communication along the edges* since only data from the immediate predecessor/successor in the tree is used.
- A sophisticated analysis could also identify the *recursion depth* as $log_2 n$, but this is at the edge of our plan's scope and also depends heavily on the shape of the tree.

From these parameters we can derive a good or even optimal initial distribution of processes and data. We do most of the computation locally and use communication to other processors only for the top level of the computation tree. With closer analysis and using the mathematical properties of the "+" operator, we might also completely unroll the lower levels of recursion into sequential summation on each processor, but this is again not our prime concern.

3.2 n-Body Problem

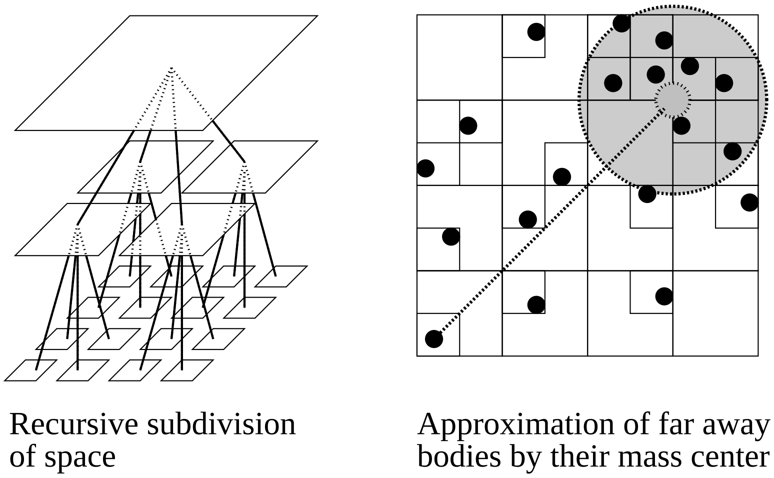

Figure 8: n-body problem: hierarchical method

This is a real-life approach that is used for galaxy simulation, molecular dynamics, etc.

In n-body calculations, we want to simulate the behavior of bodies (particles, suns, ...) under the influence of some force between the bodies (gravitation in astrophysics, electrical charges in molecular dynamics, ...). The naïve solution requires n^2 interactions between the particles since each one has to compute its influence on each other. However, we can use hierarchical methods and recursively divide space into finer regions until there are no more particles left in a region, resulting in a tree structure. Now we can cluster bodies and use their mass center for interactions with far away bodies, avoiding the complete descent in the tree and thus reducing the computation to $O(n \log n)$.

The phases of the computation – tree building, mass center computation, the time consuming force computation, and the position update – are all recursive, so we can apply our methods here.

3.3 Divide-and-Conquer

Divide-and-Conquer is a general structure for computation that follows this scheme:

```
PROCEDURE DaC(p):s =
BEGIN
  IF (IsSmall(p)) THEN      (* End of recursion            *)
    RETURN Solution(p);     (* Compute elementary solution *)
  ELSE
    q[] =  Divide(p);       (* Generate subproblems        *)
    r[] =  DaC(q[]);        (* Solve them                  *)
    RETURN Combine(r[]);    (* Combine solutions           *)
  END;
END DaC;
```

For our system, we only handle pure divide-and-conquer, i.e., current computations depend only on existing data – the problem is not inherently sequentialized by data dependencies.

We now take a look at two classical divide-and-conquer algorithms:

Quicksort – this algorithm is inherently parallel since the sorting of the subarrays can be done independently, but it is difficult to get good performance because the irregular and data dependent behavior makes load balancing difficult.

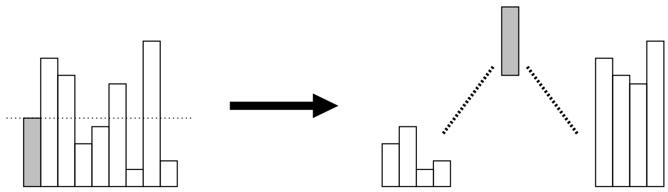

Figure 9: Quicksort: Pivot selection and division into the elements smaller and larger than the pivot element.

Matrix Multiplication – Two large matrices are multiplied by recursively subdividing them until we have a simple vector multiplication left. The resulting scalars are combined to give the result matrix.

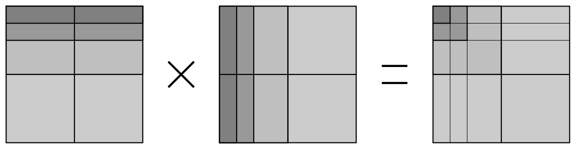

Figure 10: Matrix Multiplication: Subdivision and recombination.

The interesting point here is that all can be allocated statically for each given matrix size, because we have a regular computation that we can easily detect and completely unroll in advance. Of course for nearly every parallel computer there exists a hand-optimized library that handles problems such as matrix multiplication as efficiently as possible, but it is still interesting how well the compiler performs on this example.

4 Related Work

There are some other approaches that deal with runtime information feedback, dynamic recursive problems, or both:

In *Vienna FORTRAN's P^3T system* [Fah94], performance estimation tools are used to automatically guide compiler optimizations. Parameters are work distribution, cache behavior, and network behavior. The program is only instrumented and measured once, and the effects of all following optimizations are extrapolated from these data using machine-dependent weights. The system is limited to static array-based problems.

There are several *dedicated n-Body Programs* [Sin93] that use specialized code with explicit parallelization. While the techniques to gain speed here are not for general use, they show what the optimal compiler could produce and are therefore interesting for performance comparisons. The SPLASH benchmark suite [SWG92] for distributed memory computers contains many real-life applications, some of which can be converted for our system and used as examples.

For the *FORTRAN D* system, [CvHMS92, vHKS93, vH92] examine techniques to facilitate the parallel execution of FORTRAN programs. They concentrate on efficient communication with data prefetch and redundancy elimination and gather the necessary data with inspector/executor loops. There is also a runtime system that performs spatial (value-dependent) data placement for finite element applications.

The *Beeblebrox* system [Pip94] gives us a parallel divide-and-conquer library with dedicated C++ constructs for common cases and a runtime system.

Dynamic SPMD data structures are researched by [RCRH94, Hen90]. They use C extended with "future" calls that cause thread migration to extract parallelism from a given program, preferably automatically. Their analysis technique in the compiler tries to find divide-and-conquer computation phases and exploit them accordingly.

Finally, there is a *divide-and-conquer algebra* by [Mou90], who uses a functional parallel language to express problems such as reduction, communication, and Gaussian elimination. No larger problems are mentioned, and the efficient execution is limited to hypercube-like networks.

5 Conclusion and future work

We have shown the design of a system that can efficiently execute recursive programs in parallel.

Currently, we are working on the benchmark suite and expect to start work on the implementation of compiler and runtime system soon.

It may be necessary to narrow the scope of our approach even further – tree search algorithms, for example, also fall into the area we described here but might need still more techniques to solve them efficiently – since the time for implementation and validation is limited by the extend of the author's scholarship. There are also some practical problems to solve, e.g., the parallel machine we wanted to implement our system on is not supported by Modula-3 and porting the language would exceed our resources by far.

Despite these problems, we are confident that we will be able to present concrete results soon.

References

[CvHMS92] Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelization strategies for a molecular dynamics program. In *Technology Fucus Conference, Timberline Lodge 1992*, Houston, April 5–7 1992. intel University Partners Program, Supercomputer Systems Division.

[Fah94] Thomas Fahringer. Using the P3T to guide the parallelization and optimization effort under the Vienna Fortran compilation system. Technical Report TR 94-5, University of Vienna, April 1994.

[Hei93] Ernst A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for explicitly parallel problem-oriented programming. In *Joint Symposium on Parallel Processing*, pages 269–276, Waseda University, Tokyo, May 17–19 1993.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, Ithaca, New York, April 1990.

[Mou90] Zhijing George Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization.* PhD thesis, Yale University, May 1990.

[Phi93] Michael Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* PhD thesis, University of Karlsruhe, November 1993.

[Pip94] Andrew James Piper. *Object-Oriented Divide-and-Conquer for Parallel Processing.* PhD thesis, Emmanuel College, Cambridge, July 1994.

[RCRH94] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed memory machines. TechReport 447, Princeton University, 1994.

[Sin93] Jaswinder Pal Singh. *Parallel Hierarchical N-Body methods and their implications for parallel processors*. PhD thesis, Stanford University, February 1993.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-92-526, CSL, Stanford University, Stanford, CA, June 1992.

[vH92] Reinhard von Hanxleden. Compiler support for machine independent parallelization of irregular problems. Thesis Proposal, November 1992.

[vHKS93] Reinhard von Hanxleden, Ken Kennedy, and Joel Saltz. Value-based distributions in Fortran D – a preliminary report. Technical Report CRPC-TR93365-S, Rice University, Center for Research on Parallel Computation, Houston, TX, December 1993.