Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs

Bernd Freisleben and Thilo Kielmann

Department of Electrical Engineering and Computer Science (FB 12)
University of Siegen
Hölderlinstr. 3
D–57068 Siegen
Germany
Tel.: (49)–271–7403268
Fax: (49)–271–7402532
Email: {freisleb, kielmann}@informatik.uni-siegen.de

Abstract

Divide–and–conquer algorithms obtain the solution to a given problem by dividing it into subproblems, solving these recursively and combining their solutions. In this paper we present a system that automatically transforms sequential divide–and–conquer algorithms written in the *C* programming language into parallel code which is then executed on message–passing multicomputers. The user of the system is expected to add only a few annotations to an existing sequential program. The strategies required for transforming sequential source code to executable binaries are discussed. The performance speedups attainable will be illustrated by several examples.

Keywords:

divide–and–conquer algorithms, automatic parallelization, parallelizing compilers, message–passing multicomputers, transputer systems, workstation clusters

Abbreviated title:

Parallelization of Divide–and–Conquer Algorithms

1 Introduction

The generation of efficient parallel code for message–passing multicomputer systems is a difficult problem which involves a number of tasks that have no analogue in uniprocessor architectures. In particular, the work to be done must be partitioned into concurrently executable units, the resulting units must be mapped to the available processors, and provisions have to be taken to support interprocessor communication and synchronization. Two different approaches are currently being investigated: a) an approach where the programmer uses special parallel programming languages or language features to indicate sections for simultaneous execution [5, 7, 12, 14, 26]; and b) an approach where a program written in a sequential language is automatically transformed into a version which can be scheduled for parallel execution [15, 16, 18, 37]. In the first approach, a programmer is potentially capable of designing algorithms that may exploit parallelism better than in the second approach. This depends on the programmer's ability to come up with a clever parallel solution, the power of the parallel language features provided and the programmers's skills to use them appropriately. However, a parallel program developed in this manner is quite often intimately related to the particular multi–processor architecture considered and thus not easily portable to other parallel processors, because the programmer typically makes use of the properties of the architecture when designing the solution for a given problem. Therefore, a completely different approach may be required when the program is supposed to run on another system. Furthermore, since programmers are expected to redesign a possibly large number of already existing programs from scratch, the second approach is economically much more attractive, because it circumvents the program development overhead and avoids the portability problems associated with the first approach.

In this paper we present a system that performs automatic partitioning, mapping and communication code generation for sequential divide–and–conquer algorithms written in the *C* programming language. The divide–and–conquer paradigm is an ideal candidate

for automatic parallelization, because the philosophy behind it is to recursively fragment a problem into independent subproblems of the same kind until the problem size has been reduced to a predefined value, then solve these subproblems independently and finally combine the solutions of the subproblems [32]. Thus, the partitioning method is inherently determined by the properties of the algorithm and therefore considerably simpler than for an arbitrary sequential program. The restriction to divide–and–conquer algorithms clearly limits the applicability of our system, but considering the importance of the paradigm in a wide range of problem domains, such as sorting (quicksort, mergesort), searching (binary search), computational geometry (convex hull, closest points), graph theory (traveling salesman, depth first search) and numerical computation (matrix problems, fast fourier transform) [6, 8, 10, 11, 29, 32], the system is nevertheless extremely useful. In contrast to the efforts required to design and implement parallel divide–and–conquer algorithms by hand [19, 23, 35], the user of our system is expected to add only a few annotations to an existing sequential program in order to indicate his or her desire to run the program on a parallel architecture. Another approach aimed at easy to program parallel divide–and–conquer algorithms is the Beeblebrox system [28]. In this system, an object–oriented framework of C++ classes is provided to support the formulation of divide–and–conquer algorithms which will then be executed in parallel by a specific runtime system.

Our system generates parallel code which is then automatically downloaded and executed on a multicomputer. Currently, two hardware platforms are supported: a MEIKO transputer system [24] and a workstation cluster using the "parallel virtual machine" (PVM) software package [13]. The system's design and implementation will be discussed and runtime measurements for several example algorithms will be presented in order to illustrate the performance speedups obtained.

The paper is organized as follows. In section 2 we describe the general approaches taken to support partitioning, mapping and communication code generation. Section 3 discusses the program annotations introduced for generating parallel code. In section 4 we explain the architectural properties of our system and the strategies employed for transforming

the sequential program into a parallel version. In section 5 the implementation of the system is discussed and performance results for selected examples of divide–and–conquer algorithms are presented. Section 6 concludes the paper and outlines areas for future research.

2 Parallelizing Divide–and–Conquer Algorithms

The parallel architectures on which our work is based are message–passing multicomputers. A multicomputer is a network of processors, each having a local memory and facilities for sending messages to and receiving messages from other processors in the network; each processor has access only to its local memory, and there is no common global memory. The topology of the network may be arbitrary, but is usually organized as some well–known interconnection structure.

In such an environment there are several options for solving the problems of partitioning, mapping and communication code generation. The general approaches that we have taken are discussed in the following subsections.

2.1 The Partitioning Problem

The notion of *automatic parallelization* is usually used in conjunction with research projects on parallelizing numerical applications written in FORTRAN dialects or languages with comparable computation models [3, 9, 20, 33, 36].

Such parallelizing systems exploit *data parallelism*. Hence, they simultaneously execute operations on regularly shaped data structures like arrays or matrices. Usually, the data structures are spread over the processing elements. The generated parallel programs follow the SPMD paradigm (*single program, multiple data*) [2] where each processor executes the program on its data partition. Access to remote data items is realized by automatically inserted message passing code.

In this environment, data partitions to be distributed among the processors are determined by user interaction or by program analysis of loops operating on the data structures involved. The interferences between data partitioning, its mapping to the processors, and the generated parallel programs are crucial for determining the performance of the parallelized application programs.

Divide–and–conquer algorithms eliminate the need for developing a sophisticated partitioning technique, because the information how to partition is included in the algorithm itself. Since the initial problem is recursively divided into independent subproblems the solutions of which are again recursively combined to yield a solution for the initial problem, a divide–and–conquer computation can be viewed as the process of expanding and shrinking a tree structure. The tree is dynamically formed during the computation; whenever a new subdivision is performed within the algorithm, descendants of a node will be created. Additional techniques to discover concurrently executable units, as required for an arbitrary sequential program, are not necessary.

The generated parallelized programs call the divide–and–conquer routines on remote processors while exchanging parameters and results. This leads to a program structure known from *functional parallelism* [7], where the set of functions (routines) of a given program is distributed over the processors while each processor services requests for execution of its routines. In the special case of divide–and–conquer algorithms it is one unique function which is placed on the processors. Thus, the divide–and–conquer paradigm leads to SPMD programming, too, and furthermore constitutes the junction between data parallelism and functional parallelism.

2.2 The Mapping Problem

The mapping problem involves the decision of how the concurrently executable units determined by the partitioning method are assigned to the processors of the multicomputer system in order to achieve a maximal degree of parallelism in an efficient manner. Since

the mapping will affect the overall execution time and thus the speedup attainable by the parallel program, it must be chosen carefully. In particular, it should keep the computational load of the processors balanced and should minimize the communication overhead among the processors. Because of its importance, the mapping problem has attracted a great amount of research efforts during the last years [27].

Systems exploiting the data parallel approach like [3, 9, 20, 33, 36] develop mapping strategies which originate from a given data distribution. The goal of such systems is to maximize *locality* of data accesses [31]. In this respect, every processor should execute a partition of the program which allows it to operate mainly on the data located in its local memory. Thus, finding suitable data distributions for given multicomputer architectures is a difficult task for data parallel systems and hence therefore usually relies on user interaction.

The concept of locality of data accesses is not directly applicable to divide–and–conquer parallelization, because the parallelized programs are structured in a different manner, as described above. Nevertheless, it is still extremely important to minimize the overall communication costs.

A divide–and–conquer algorithm accesses sets of data which are defined by the algorithm's parameter and result declarations. Due to the completely dynamic nature of the call relation between the processors, it is impossible to pre–distribute data at compile time. Instead, the data necessary for computing the whole problem is located on the processor assigned to it. Data sets for computing subproblems are passed to another processor along with the corresponding function call.

A proper definition of parameters and results yield a minimal set of data to be communicated between the processors. Thus, minimizing parameter data is the divide–and–conquer equivalent to maximizing locality. The minimal size of parameter and result data is a property of the algorithm to be parallelized and thus determines the algorithm's suitablility for parallelization. As a consequence, mapping strategies for divide–and–conquer

algorithms try to achieve different goals.

Several approaches for mapping divide–and–conquer algorithms to multiprocessor architectures have been proposed in the literature [23, 35], and similar to these proposals we also simplify the following discussion by assuming that the divide–and–conquer paradigm is based on two recursive subdivisions of the problem instead of the n subdivisions which are possible in general. An obvious approach is to let one processor divide the problem into two subproblems and pass these on to two further processors. This procedure is recursively continued until the problem has been broken up suffiently, resulting in a mapping to an architecture designed as a binary tree topology. The processors at the leaves of this tree solve the subproblems assigned to them and propagate the results to their parents. These in turn combine the results and send them to their parents until the processor at the root is reached.

The binary tree approach has two major drawbacks. First, the number of processors required must be a power of 2 (minus 1) and second, many of these processors (all interior nodes in the tree) are idle while the leaf processors perform the computation, because they are only involved in dividing the subproblems and combining their solutions.

Another approach which avoids these drawbacks [23] is to let one processor divide the initial problem into two subproblems, pass one of these to a further processor and keep the other half to itself. Every processor repeats this step recursively until the problem size is sufficiently small and performs the computation assigned to it, which effectively constitutes a mapping to a binomial tree topology [34]. The results are propagated up the tree and combined in the reverse order in which the subproblems were passed down the tree. It is intuitively clear that the number of processors required in this topology is less than in a binary tree and that a processor, after having received a subproblem, only idles when its computation is finished or the problem sizes are significantly imbalanced. However, the particular binomial tree topology required for a given algorithm cannot be wired up in advance, because its structure is data dependent and therefore highly irregular.

As a consequence, it is better to adopt the binomial tree as the basic computation graph and map it to more regular physical topologies. This is exactly the mapping approach that we have used in our system, and the physical topologies investigated are the hypercube interconnection structure, the cube–connected–cycles network [25], and an ethernet–based bus system.

2.3 Communication Code Generation

The fundamental communication pattern used in our system is based on a master/worker organization in which a single master task distributes the work to a set of worker tasks via an asynchronous remote procedure call mechanism. The master/worker principle is applied recursively in our approach for each of the subproblems created during the divide–and–conquer computation. In addition to the initial master task, which is responsible for the particular problem to be solved, there is a unique *scheduler* task which controls the pool of available processors and is thus the only entity that knows the physical topology of the network. The scheduler is responsible for performing the mapping from the binomial tree to the underlying physical topology during runtime. The initial master and the scheduler are assigned to the same dedicated processor, while each worker runs on a unique processor contained in the pool.

Initially, both the master and all workers identify themselves to the scheduler. The problem to be solved is then given to the master who in turn divides it and requests the unique identification of an idle worker from the scheduler. The request is serviced by returning the identification of the worker residing on the nearest idle processor in accordance to the physical topology. Upon receipt, the master sends the subproblem to this worker. The worker repeats the process and thus acts as the master of the subproblem which it intends to propagate. Once a worker has finished its computation and has returned the results to its master, the master informs the scheduler to release the worker and make it again available to the pool of idle processors. The whole computation is finished when the master terminates and all workers have been returned to the pool.

It is important to find a set of subproblems to be solved concurrently which is optimal in order to minimize completion time. Trying to propagate all subproblems leads to optimal load balancing between the processors because there will be unsolved problems at almost every time of the computation. But processor utilization is far from optimal because of prohibitive communication overheads.

The opposite strategy for assigning subproblems to processors, well known from parallelizing compilers for FORTRAN dialects, performs this mapping statically at compile time in order to minimize communication overhead. Unfortunately, this approach is not feasible for divide–and–conquer parallelization, because the set of subproblems to be solved depends on the actual parameters of the divide–and–conquer algorithm.

A nearly optimal solution to this problem can be obtained by applying a technique called *dynamic partitioning*. In this approach, also used in [16, 30], it is decided at runtime whether a given subproblem should be propagated to another processor or better be computed locally.

In our particular model, every processor computes exactly one problem at a time. In order to utilize n processors, communication overheads become minimal when only the n computationally "largest" subproblems are spread over the processors. Thus, every processor computes exactly one subproblem. Assuming that the completion time of a subproblem is a function which monotonically increases with the problem size, our experiments have shown that the completion time can be minimized when subproblems are propagated to other processors if the following inequality is true:

$$S_c \times f > S_i / n$$

where n is the number of available processors, S_i is the initial (whole) problem size, S_c is the size of the problem currently considered, and f is the factor determining in how many subproblems (usually 2) a problem is divided.

3 Program Annotations

The system presented in this paper processes the *C* language conforming to the ANSI standard [17], but a few program annotations are required for generating code which is able to operate in parallel. These annotations are only of a declarative nature, included to overcome *C*'s deficiencies related to the description of formal parameters and side effects in functions. Thus, the annotations are not necessary as far as partitioning and mapping for parallel execution is concerned. The two main problems which cause the need for program annotations are the following:

1. Unfortunately, *C*'s declaration of formal parameters is far from being as complete as it would be necessary to transport parameters and results between different processors.

   First, the parameter declaration does not express which parameters are read by a function ("input parameters") and which ones will be modified ("output parameters"). In the presence of further function calls from inside the function's body, it is also impossible to derive this information from the statements being executed.

   Second, the parameter declaration provides no information about the memory areas used to hold the set of parameters. In particular, the notations for arrays and pointers together with pointer aliases cause intractable problems. It has been proven in [21] that in languages like *C* which allow arbitrary pointers, the problem of determining aliases at a program point is $\mathcal{P}$-space-hard.

2. In our system, recursive function calls are performed using an asynchronous remote–procedure–call ("RPC") mechanism. Its asynchronous nature implies the need for a synchronization mechanism — the caller has to wait for the callee's termination before it may use its results.

   Again, in the presence of further function calls from inside the function's body, it is impossible to derive which statements access the results of a callee. Therefore, the

programmer has to indicate a "synchronization point" after which he or she expects all results to be available. This conforms to the implementation of an asynchronous RPC suggested in the literature [22].

The program annotations introduced are explained in detail in the following:

parallel This new keyword preceeds a function definition and simply tells the compiler that this function is intended to be executed in parallel.

sync This newly introduced statement defines the "synchronization point" after which all results of recursive subproblems are expected to be available.

input The set of input declarations (located between the parameter declarations and the function's body) describe which parameters have to be provided to a (possibly remotely) called function. If necessary, these declarations also provide information about array sizes which may be stated in terms of parameter variables or constants.

output Analogously to *input*, these declarations (together with the function's return value) form the set of a function's results.

problemsize This declaration is used to denote a numerical value describing the size of the problem currently dealt with by an invocation of the function, particularly in relation to recursive calls for subproblems. This declaration is only needed for the dynamic partitioning technique as discussed in section 2.3.

We will now illustrate these annotations by using the well known *quicksort* algorithm as an example. Figure 1 shows the C–code with all additional declarations. They are printed in *italics* to distinguish them from the rest of the code.

please insert fig. 1 here

It is easy to see that this function is intended to execute in parallel, sorts `n` + 1 elements per invocation, and reads the parameter `n` as well as `n` + 1 elements of `array`. The latter ones will also be modified. The synchronization point forms the end of the function's body, because it is not necessary to combine the sorted subarrays.

4 System Description

In this section the functionality of our system for automatic parallelization of divide–and–conquer algorithms is presented. In particular, we describe all the steps required for transforming a sequential program, available as *C* source code, into concurrently excutable binaries. Figure 2 shows these steps graphically. The naming conventions for files and programs are identical to those common in a UNIX environment. The individual programs performing the transformation steps are listed at the bottom of the figure.

please insert fig. 2 here

4.1 Preprocessing

In the first step, the original *C* source code is precompiled with the C–preprocessor (`cpp`) in order to allow a parser to operate on the program, because the preprocessor supplies all declarations from included files and transforms the input into a version which conforms to the ANSI *C*–syntax defined in [17].

The C–preprocessor is also used to allow conditional compilation for either sequential or parallel execution. The reader may have observed in the previous section that the syntax of our program annotations is compatible with the C–preprocessor's macros. The programmer has to include a special header file containing macro definitions which conditionally remove all annotations and thus allow (traditional) compilation for sequential execution.

4.2 Restructuring the Source Code

After the preprocessing step has been performed, `dcpar` restructures the source code into three parts:

1. The "master part" contains the original `main()` function, which will become part of the master program. To resolve name conflicts, it will be renamed and can thus

be called from the master's `main()` function.

2. The "function part" is used for both the master and the worker program. It replaces the recursive function calls and directs them to a stub. Within this stub it is decided whether the function will be executed on another processor. If not, the call is directed to the original function. Additionally, code for transmitting parameters and receiving results is located in this part.

3. The "worker part" contains code for receiving the parameters, calling the function and returning the results.

We will now take a closer look at the key transformation which is performed on a recursive function. Let us assume that there is a function `f` with a single parameter `a` of type `T1` and a return value of type `T2`:

```
parallel T2 f (T1 a)
   input(a)
{
   ...
   result = f(a1) + f(a2);
   ...
   sync;
   return (result);
}
```

The function part (as mentioned above) will contain the following function, in which the original recursive call is divided into calls to the stub and the local function (for the last subproblem) and the final combination of the results. An array is automatically generated which temporarily stores the subproblem's results:

```
T2 local_f (T1 a)
{
   T2 result_vector[2];
   ...
   stub_for_f (&result_vector[0],a1),
   result_vector[1] = local_f(a2);
   ...
   result = result_vector[0] + result_vector[1];
   return (result);
}
```

4.3 Compiling and Linking to Executable Programs

The master- and function parts produced so far are then concatenated (by `cat`) in order to form the application specific part of the master task described in section 2.3. The same procedure is applied to the worker- and function parts to obtain the application specific part of the worker task.

Afterwards, these parts are separately compiled with the *C* compiler of the target system (by `cc`), resulting in two object files.

In the last step, the object files produced so far are linked together (via `ld`) with the appropriate `main()`–programs and a hardware–architecture specific "communications library".

This library contains the implementation of an asynchronous remote-procedure-call mechanism. It consists of several modules one of which realizes the interface to the underlying runtime system of the target machine, which in our case are the runtime systems of the MEIKO Computing Surface transputer network [24] and the PVM package [13].

The main program for the master initializes the communication modules and calls the (now renamed) function `main()` from the original source code, passing through its command line parameters. When the computation is finished, the master terminates the worker tasks.

The main program for the workers also initializes the communication modules and then waits for incoming messages indicating requests for calling the recursive function. After finishing a computation, a worker returns the results and waits for the next request. If a worker receives a message to terminate, it stops running.

5 Implementation and Performance

Two message passing multicomputer systems were used to run the programs in parallel. The first is the MEIKO Computing Surface [24], consisting of 72 transputers T800 (25 MHz) with 1 MB RAM and a SUN 4/390 acting as the host computer. The second is a set of 12 SUN Sparcstation 10/20 workstations, interconnected via a local ethernet cable and communicating using the PVM software package [13].

These two architectures have been chosen to investigate the runtime behaviours in a tightly coupled system with fast communication (transputers) as well as in a situation in which communication is very expensive but the pocessors are more powerful (workstation cluster).

5.1 Example Algorithms

In order to investigate the performance of our system, we have programmed several divide–and–conquer algorithms and measured their runtimes in comparison to the sequential execution of the programs. The following examples of divide–and–conquer algorithms have been selected:

1. The first example is the well–known quicksort algorithm [11] for sorting an array of n integer numbers.

2. The second example is an algorithm that multiplies two matrices of size $n \times n$. The algorithm is based on an iterative version consisting of three nested loops and

requiring $O(n^3)$ multiplications. The outermost loop has been replaced by recursive calls for two multiplications of size $n/2$.

3. The third example is an algorithm for adaptive numerical integration [4], i.e. computing $\int_a^b f(x)dx$ for a function $f(x)$ in the interval between the points a and b. The basic idea of the algorithm is that $f(x)$ is replaced by a straight line from $(a, f(a))$ to $(b, f(b))$ and the integral is approximated by computing the area bounded by the resulting trapezoid. This process is recursively continued for two subintervals as long as the area differs significantly from the sum of the areas of the subintervals' trapezoids. The algorithm is adaptive in the sense that the partitioning into subintervals depends on the shape of the function considered.

4. The fourth example is an algorithm for solving the *knapsack problem* [32], which is defined as the problem of finding a set of items each with a weight w and a value v in order to maximize the total value while not exceeding a fixed weight limit. The problem for n items is recursively divided into two subproblems for $n-1$ items, one with the missing item put into the knapsack and one without it.

5. The fifth example is the MERGEHULL algorithm [29] for determining the convex hull of n points in a plane. This algorithm divides the original set of points S into two subsets S_1 and S_2 of approximately equal cardinalities, recursively computes their convex hulls $CH(S_1)$ and $CH(S_2)$ and combines them using a Graham scan to form $CH(S)$.

6. The sixth example is an algorithm for the decomposition of positive integer numbers into prime factors, also known as *integer factorization* [11]. The algorithm implemented here is not one of the popular heuristic ones. Instead, it finds all prime factors of a given number N in the interval ranging from n_1 to n_2. In order to achieve this, it divides the given interval into two of half the size and tests their elements respectively.

7. The last example is an exact algorithm for the set–covering problem [11]. The task is to find a minimal set of subsets of a set S which covers the properties of all elements of the set S.

5.2 Performance Evaluation

The seven programs have been automatically parallelized with our system and executed on both multicomputers. On the MEIKO transputer system, the physical topologies used were hypercubes for $p = 2$, $p = 4$ and $p = 8$ processors and cube–connected cycles (with 2 processors in each corner) for $p = 16$ and $p = 32$. The latter topology has been selected to minimize the path length under the constraint that each transputer is equipped with four physical links only. For $p = 32$ it is illustrated in Figure 3. The interconnection topology of the workstation cluster is simply a bus system due to the ethernet cabling.

please insert fig. 3 here

The programs have also been compiled to run sequentially on one processor. The runtimes measured for the sequential programs T_s were used to compute the speedup $S = T_s/T_p$, where T_p is the runtime of the parallel execution with p processors. The speedup factors achieved on the transputer system are shown in Figure 4, the ones obtained on the workstation cluster can be found in Figure 5.

please insert figs 4 and 5 here

The speedups achieved for the individual problems differ significantly and thus indicate the suitability of the different algorithms for parallel execution in a message–passing multicomputer environment. Since in the absence of a shared memory the total costs are clearly dominated by communication, we will evaluate the measured runtime behaviours based on the algorithms' computational time complexity C and the parameter data size P, as introduced in section 2.2. The following table lists these properties of the selected algorithms; n denotes the problem size.

Algorithm	Time Complexity C	Parameter data size P
Quicksort	$O(n \log n)$	$O(n)$
Convex Hull	$O(n \log n)$	$O(n)$
Matrix Multiplication	$O(n^3)$	$O(n^2)$
Adaptive Integration	$O(n)$	$O(1)$
Integer Factorization	$O(n\sqrt{n})$	$O(1)$
Set Covering	$O(2^n)$	$O(n^2)$
Knapsack	$O(2^n)$	$O(n)$

As shown by Figures 4 and 5, the example algorithms can be divided into two groups: algorithms with or without reasonable speedups. The latter group consists of quicksort, convex hull, and matrix multiplication. These algorithms have a low ratio of C/P in common, differing only by constant factors. On the transputer system, where communication is relatively cheap, matrix multiplication and convex hull show small speedups. But on the workstation cluster with fast processors and a slow network, they do not exhibit considerable speedup factors.

The group of algorithms with reasonable performance, namely adaptive integration, integer factorization, set covering, and knapsack are characterized by a high ratio C/P, so computation time saved by parallel execution is not outweighed by communication costs. The program for adaptive integration shows a good speedup behaviour although its C/P ratio does not look better than the ratio for matrix multiplication. This is essentially due to the very small, constant communication cost of only three floating–point values per function call.

Summarizing our results, the divide–and–conquer paradigm has a good potential for parallel applications while a particular algorithm's suitability for parallelization can directly be derived from its C and P parameters.

6 Conclusions

In this paper we have presented a system that automatically partitions sequential divide–and–conquer algorithms programmed in C into independent tasks, maps these to multi-

computer systems and executes them in parallel. The user is required to make only minor modifications to an existing sequential program in order to tell the system that it should parallelize the algorithm. Apart from using standard UNIX tools and compilers for the parallel target machines, the system is fully implemented in *C* and thus easily portable to other architectures. Several examples of divide–and–conquer algorithms have been parallelized in order to measure the speedups attainable. It has been shown that these speedups are primarily dependent on the properties of the algorithms, such as computation time complexity and parameter data size.

There are a number of issues for future research, such as exploiting knowledge about algorithm properties for parallel–code generation, developing alternative execution models for parallel divide–and–conquer algorithms based on a virtual shared memory, and investigating the automatic parallelization of further classes of algorithms.

References

[1] ACM SIGARCH AND IEEE COMPUTER SOCIETY. *Supercomputing '93*, Portland, Oregon, IEEE Computer Society Press, 1993.

[2] G. S. ALMASI, A. GOTTLIEB. *Highly Parallel Computing.* Benjamin/Cummings, 1994.

[3] F. ANDRÉ, M. LE FUR, Y. MAHÉO, J.-L. PAZAT. The Pandore Compiler: Overview and Experimental Results. Technical Report No. 869, IRISA, Cedex, France, 1994.

[4] G. R. ANDREWS. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[5] G. R. ANDREWS, R. A. OLSSON. *The SR Programming Language.* Benjamin/Cummings, 1993.

[6] Z. BAI, J. DEMMEL, M. GU. Inverse Free Parallel Spectral Divide and Conquer Algorithms for Nonsymmetric Eigenproblems. Research Report 94–01, Department of Mathematics, University of Kentucky, USA, 1994.

[7] H. E. BAL, J. G. STEINER, A. S. TANENBAUM. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[8] C. BISCHOF, X. SUN. A Divide–and–Conquer Method for Computing Complementary Invariant Subspaces of Symmetric Matrices. Technical Report MCS–P286–0192, Argonne National Laboratory, 1992.

[9] Z. BOZKUS, A. CHOUDHARY, G. FOX, T. HAUPT, S. RANKA. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In *Supercomputing '93* [1], pp. 351–360.

[10] G. BRASSARD, P. BRATLEY. *Algorithmics—Theory and Practice.* Prentice Hall, 1988.

[11] T. H. CORMEN, C. E. LEISURESON, R. L. RIVEST. *Introduction to Algorithms.* MIT Press, 1990.

[12] I. FOSTER, S. TAYLOR. *Strand—New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[13] G. A. GEIST, A. L. BEGUELIN, J. J. DONGARRA, W. JIANG, R. J. MANCHEK, V. S. SUNDERAM. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[14] A. S. GRIMSHAW. Easy–to–Use Object–Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39 – 51, 1993.

[15] G. GUPTA, V. S. COSTA. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In D. ETIEMBLE, J.-C. SYRE (Eds.), *PARLE'92 Parallel Architectures*

and Languages Europe, Paris, France, Lecture Notes in Computer Science, No. 605, pp. 617 – 634, Springer-Verlag, 1992.

[16] W. L. HARRISON III, Z. AMMARGUELLAT. The Design of Automatic Parallelizers for Symbolic and Numeric Programs. In T. ITO, R. H. HALSTEAD, JR. (Eds.), *Parallel Lisp: Languages and Systems*, Sendai, Japan, Lecture Notes in Computer Science, No. 441, pp. 235 – 253. Springer-Verlag, 1989.

[17] B. W. KERNIGHAN, D. M. RITCHIE. *The C Programming Language*, 2nd edition. Prentice Hall, 1988.

[18] C. W. KESSLER (Ed.). *Automatic Parallelization.* Vieweg-Verlag, 1994.

[19] R. KNECHT. Implementation of Divide-and-Conquer Algorithms on Multiprocessors. In J. D. BECKER, I. EISELE, F. W. MÜNDEMANN (Eds.), *Parallelism, Learning, Evolution*, Neubiberg, Germany, pp. 121–136, Springer-Verlag, 1989.

[20] C. KOELBEL, P. MEHROTRA, J. V. ROSENDALE. Supporting Shared Data Structures on Distributed Memory Architectures. *SIGPLAN Notices*, 25(3):177–186, 1990.

[21] W. LANDI. *Interprocedural Aliasing in the Presence of Pointers.* PhD Dissertation, Technical Report LCSR-TR-174, Rutgers University, USA, 1992.

[22] B. LISKOV, T. BLOOM, D. GIFFORD, R. SCHEIFLER, W. WEIHL. Communication in the Mercury System. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, 1988.

[23] V. M. LO, S. RAJOPADHYE, S. GUPTA, D. KELDSEN, M. MOHAMED, J. TELLE. Mapping Divide-and-Conquer Algorithms to Parallel Architectures. In *International Conference on Parallel Processing*, Vol. III, pp. 128–135, 1990.

[24] MEIKO LIMITED. *Meiko Computing Surface, CS Tools Documentation*, 1989.

[25] B. MONIEN, H. SUDBOROUGH. Embedding one Interconnection Network in Another. In G. TINHOFER, E. MAYR, H. NOLTEMEIER (Eds.), *Computational Graph Theory*, Computing Supplementum, No. 7, pp. 257–282, Springer-Verlag, 1990.

[26] O. NIERSTRASZ. The Next 700 Concurrent Object–Oriented Languages. In D. TSICHRITZIS (Ed.), *Object Composition*, pp. 165–187. Centre Universitaire d'Informatique, University of Geneva, 1991.

[27] M. G. NORMAN, P. THANISCH. Models of Machines and Computation for Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[28] A. J. PIPER. *Object–Oriented Divide–and–Conquer for Parallel Processing.* PhD Dissertation, Emmanuel College, Cambridge, UK, 1994.

[29] F. P. PREPARATA, M. I. SHAMOS. *Computational Geometry—An Introduction.* Springer-Verlag, 1985.

[30] F. A. RABHI, G. A. MANSON. Divide–and–Conquer and Parallel Graph Reduction. *Parallel Computing*, 17:189–205, 1991.

[31] A. ROGERS, K. PINGALI. Process Decomposition Through Locality of Reference. In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, Portland, Oregon, pp. 69–80, 1989. Published as SIGPLAN Notices 24(7), 1989.

[32] R. SEDGEWICK. *Algorithms.* Addison Wesley, 1988.

[33] C.-W. TSENG, S. HIRANANDANI, K. KENNEDY. Preliminary Experiences with the Fortran D Compiler. In *Supercomputing '93* [1], pp. 338–350.

[34] J. VUILLEMIN. A Data Structure for Manipulating Priority Queues. *Commun. ACM*, 21(4):309–315, 1987.

[35] I.-C. WU. Efficient Parallel Divide-and-Conquer for a Class of Interconnection Topologies. In W. L. HSU, R. C. T. LEE (Eds.), *ISA '91 Algorithms*, 2nd Interna-

tional Symposium on Algorithms, Taipei, Republic of China, pp. 229–240, Springer-Verlag, 1991.

[36] H. ZIMA, P. BREZANY, B. CHAPMAN, P. MEHROTRA, A. SCHWALD. Vienna Fortran – A Language Specification, Version 1.1. Technical Report Series ACPC/TR 92-4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.

[37] H. ZIMA, B. CHAPMAN. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. ACM Press, 1990.

List of Figures

Figure 1: Quicksort with Annotations for Parallel Compilation

Figure 2: Transformation of Sequential C Code to Parallel Binaries

Figure 3: Cube–Connected Cycles Topology for $n = 32$

Figure 4: Algorithm Speedup on a Transputer System

Figure 5: Algorithm Speedup on a Workstation Cluster

```
parallel void qsort (int n, int array[])
   problemsize(n+1)
   input(n)
   input(array[n+1])
   output(array[n+1])
{  int i,j,x,w;
   i = 0; j = n; x = array[n/2];
   do  {
       while (array[i] < x) i++;
       while (x < array[j]) j--;
       if (i <= j) {
          w = array[i]; array[i] = array[j]; array[j] = w;
          i++; j--;
       }
   } while ( i<=j );
   if (0<j) qsort(j,array);
   if (i<n) qsort(n-i,array+i);
   sync;
}
```

Figure 1: Quicksort with Annotations for Parallel Compilation

Freisleben/Kielmann, Figure 1

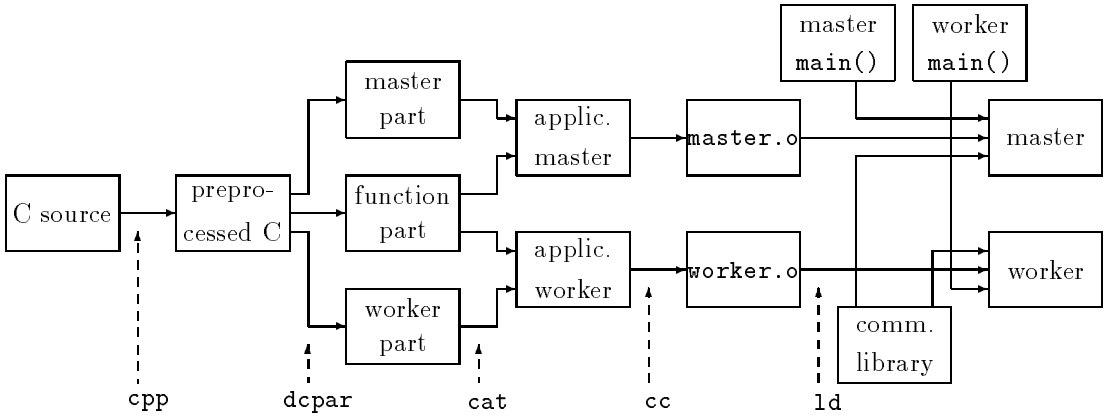

Figure 2: Transformation of Sequential C Code to Parallel Binaries

Freisleben/Kielmann, Figure 2

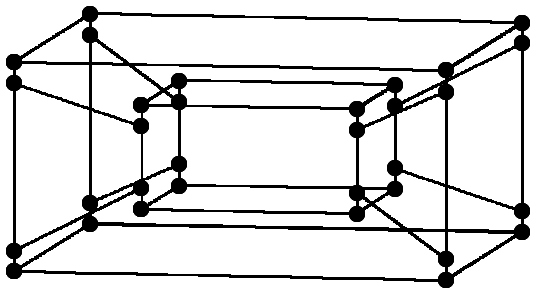

Figure 3: Cube–Connected Cycles Topology for $n = 32$

Freisleben/Kielmann, Figure 3

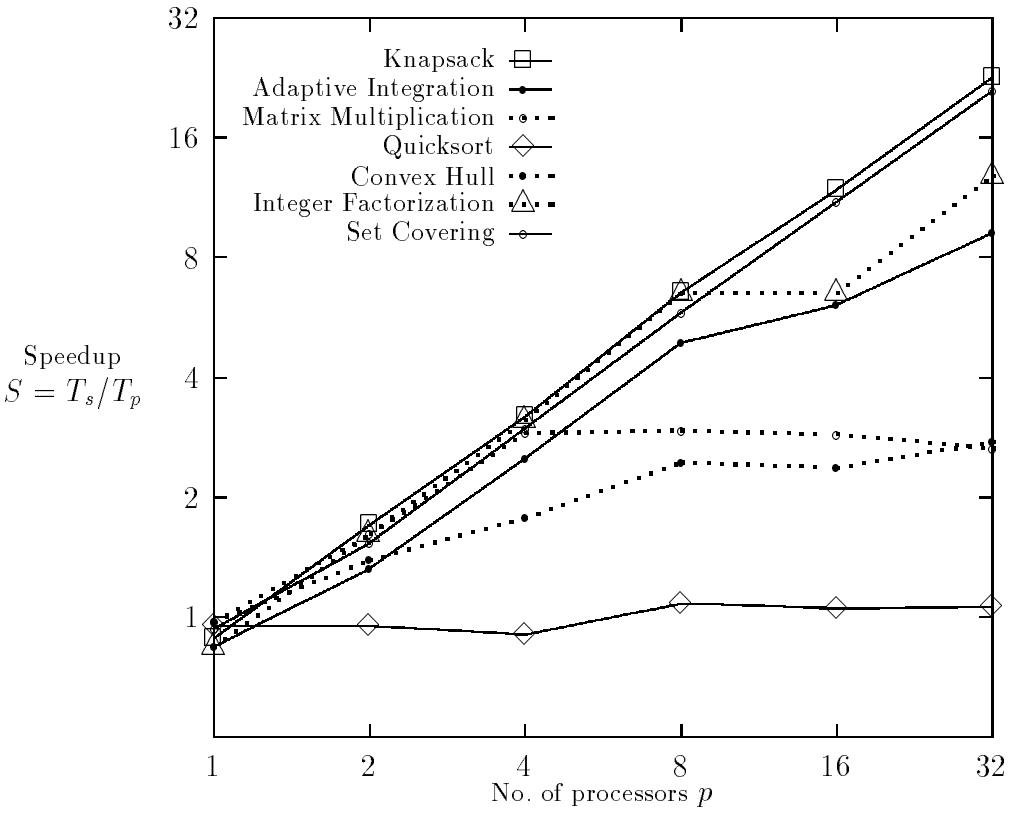

Figure 4: Algorithm Speedup on a Transputer System

Freisleben/Kielmann, Figure 4

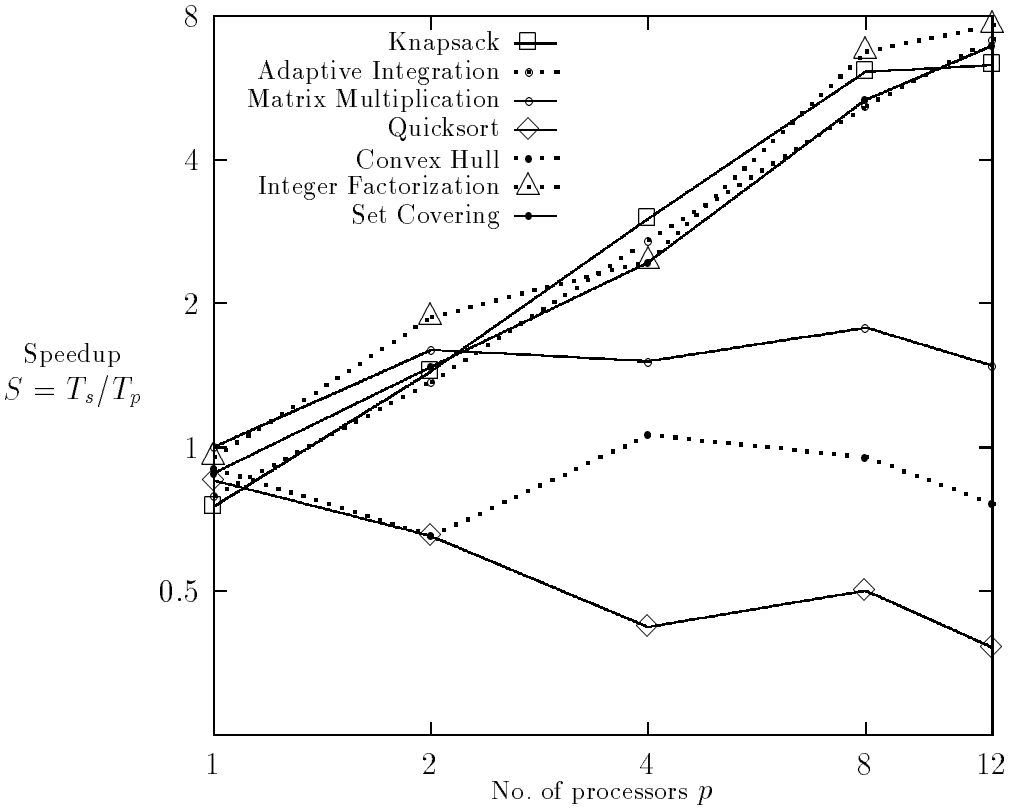

Figure 5: Algorithm Speedup on a Workstation Cluster

Freisleben/Kielmann, Figure 5