The PARAMAT Project: Current Status and Plans for the Future

Christoph W. Keßler*
Fachbereich 4 – Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: kessler@ti.uni-trier.de

Abstract

Scalable parallel numerical libraries and automatically parallelizing compilers seem to be contrary approaches to the same goal: the user–friendly generation of efficient parallel numerical programs for shared–memory and distributed–memory multiprocessors.

We propose a framework that integrates the library approach and parallelizing compiler technology. It is based on fast and powerful pattern recognition in sequential source programs and considerate local algorithm replacement.

By a simplified prototype implementation, we demonstrate the functionality of this approach for a massively parallel shared–memory target machine, the SB-PRAM.

We further propose constructive guidelines to adapt the method to distributed–memory multiprocessors by integrating an automatic array distribution engine, by synthetic performance prediction, and by data–distribution–independent design of library routine specification.

Key words: Massively parallel computer, automatic parallelization, scalable parallel library, data distribution, pattern matching, PARAMAT system, SB-PRAM, Fork95

1 Introduction

It is a well–known fact that, unfortunately, parallelizing compilers have severe problems with generating efficient code for distributed memory target machines (DMS). This is due to several reasons: First, there are the NP–complete problems of finding suitable static [LC90] and dynamic [Kre93] distributions of the source program's arrays among the processors. Second, faced with today's distributed-memory systems, it gets more and more difficult for the compiler to predict the run time of a nontrivial parallelized program, due to caching, network contention, buffering and other hardware properties. Precise and quickly accessible run time information is, however, essential for global optimization of array distributions (e.g., [BKK94]). Third, a program that has been generated by a (semi-)automatic parallelizer by adapting the sequential code to given array distributions and inserting message passing instructions where nonlocal data may be required, must, in general, be transformed to perform efficiently on a DMS. However, it is hard for the compiler to select among the numerous optimizing transformations that are known today, from simple loop transformations up to local algorithm exchange. Furthermore, the compiler should be able to exploit hardware–supported higher–order communication primitives like broadcast,

*Research was partly supported by DFG, SFB 124 TP D4, Saarbr"ucken University, Germany.

combine or reduce [LC91]. Maintaining interfaces to machine–independent message–passing libraries like P4 [BL92], PVM [LT93] or MPI [CGH94] may also be a source of inefficiency.

Thus, an access to expert's knowledge would be fine. One approach to exploit this idea has been followed during the last years with the development of parallel libraries for distributed-memory multiprocessors (e.g., [CDPW92], [FSSS92], [Thi93]; [KA93]) that can be used by the developer of a new program. We call a library of parallel routines *scalable* if the efficiency of each routine does, over a wide range, not primarily depend on the local problem granularity[1]. This will, in general, mean that in practice such a scalable routine contains several (parallel) algorithms for different magnitudes of local problem size.

Scalable libraries offer several advantages: The operands are explicitly given in a clean interface; this may make address calculations simpler. As the implemented algorithm of the library routine is known, the implementation can use explicit formulae for iteration sets, communication sets etc. [KMR90]. Overlapping communication with computation is generally possible. For library routines, the expense for assembler programming (which is highly required for many systems to get a maximum of node performance, see [Fri91]) is more profitable than for a user program which is used only once. Moreover, practically significant approaches to automatic array distribution (e.g., [BKK94]) require extensive knowledge-based compilation environments, e.g. precomputed run time tables.

However, scalable libraries for DMS have also disadvantages: First, a programmer must get used to a parallel library pack, and there has no standard been established yet. For dusty decks, the library approach is not applicable. Second, and even more important: There is no generally accepted way of how to obtain data distribution independence. Either, a library routine assumes or enforces a fixed distribution for each array operand (which makes programming of the routine easy, but severely limits efficiency), or the distributions of array operands must be passed to the routine (which makes efficient usage of the routine easy, but renders programming and interface more complicated), either explicitly as a parameter, or implicitly as a global variable. Data distribution independence will, if done right, result in a dramatical blow-up of library code size, since the parallel algorithms are usually *not* data distribution independent, and the different cases must be distinguished.

Furthermore, contemporary libraries for DMS often do not distinguish between multidimensional arrays and array access shapes like vectors or matrices; e.g., a matrix is assumed to be a two-dimensional array. We think that, although this case appears to occur frequently in practice, this is too restrictive. We rather regard a vector or a matrix as a 1D or, respectively, 2D data window onto a (potentially) higher–dimensional array.

Last but not least, such libraries are typically programmed in (message–passing–) C or Fortran, and thus the efficiency of the generated routine is limited by the quality of the compiler used, and so is the predictability of run times.

We offer an alternative to this classical library approach: We integrate the library, including extensive run time tables and handling of array distributions, into the compiler. Additionally, we offer a pattern recognition tool that fast and reliably detects occurrences of computations in the sequential source for which a parallel implementation routine exists [Keß94b] and replaces them by a call to this routine. This pattern–driven approach (named "PARAMAT", see [KP93, Keß95]) is motivated by the observation that programs that are reasonable candidates to be ported to DMS (especially, programs operating on vectors and dense matrices) are at a large degree made up of a relatively small set of typical, frequently occurring basic numerical operations ("patterns"). This observation has been supported by an empirical study of numerical

[1]We define *granularity* as the quotient of the overall problem size divided by the number of processors.

algorithms and source codes [Keß94a]. [Keß94a] also contains the complete list of patterns.

In this paper, we will make clear that the integration of library and parallelizer can be realized by our pattern–driven approach. After a short summary of PARAMAT in section 2, we propose overall guidelines for the construction of library routines (section 3), present an implementation for a shared memory target machine (section 4) and our plans for the specification of a data-distribution-independent and compiler-integrated scalable parallel library (section 5).

2 Pattern–driven automatic parallelization

Patterns are semantics–based, mostly data–parallel operations, such as e.g. elementwise vector and matrix operations, basic linear algebra operations, simple recurrences, grid stencils, relaxation operations, simple reduction operations, convolutions, shifts etc. A pattern is regarded to be a primitive with respect to mathematical properties, occurring data structures, memory access structure, array alignment preferences and run time behaviour. Regarding numerical operations on vectors and dense matrices, we have collected a list of 150 typical patterns (see [Keß94a]).

A *pattern instance* is generated if a piece of code has been recognized as an occurrence of a pattern. Unparsed, it looks similar to a call to an externally defined procedure, with the formal pattern operands ("slots") being bound to the corresponding program objects. Pattern instances can either be generated by our pattern recognition tool [Keß94b] or directly by the programmer (just bypassing the pattern recognizer).

Pattern recognition is robust against many typical code transformations encountered in sequential programs, such as loop distribution and interchange, loop blocking, loop unrolling and jamming, temporary variables. Exploiting the natural semantical interdependencies among the patterns makes the pattern recognition algorithm quite fast. More details on the pattern recognizer are beyond the scope of this report; they are published elsewhere [KP93, Keß94b, Keß94a]. For illustration, the following NAS Kernel

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= l; i++) {
        c[i][k] = 0.0;
        for (j = 1; j <= m; j += 4)
            c[i][k] = c[i][k]  +  a[i][j]   * b[j][k]
                               +  a[i][j+1] * b[j+1][k]
                               +  a[i][j+2] * b[j+2][k]
                               +  a[i][j+3] * b[j+3][k];
    }
```

is recognized by the pattern recognition tool as an instance of Matrix–Matrix–Multiplication

```
_m=4*((m-1)/4)+4;
MM (i,k,j,c[1:l][1:n],a[1:l][1:_m],b[1:_m][1:n],0.000000);
```

within 0.3 seconds on a low–end SUN SLC including lexing and parsing the source and unparsing and printing the result. Many more results are listed in [Keß94a].

Due to the pattern instances, we know what the program locally does; thus we can infer additional knowledge on mathematical properties, efficient parallel implementations, their favorable array distributions and their run time behaviour on the target machine considered. The parallelization system can then easily use this knowledge to guide a sophisticated parallelization process including high–level program transformations up to local algorithm replacement.

3 Pattern implementations: organization and format

3.1 Code driver

The general structure of a PARAMAT back-end for distributed memory target machines has been outlined in [Keß95].

To access the output of the pattern recognizer, a *driver* for code generation is required (see Figure 1). This driver unparses the abstract syntax tree that has been constructed by the pattern recognition tool, while paying attention to directives from the PARAMAT user that may prevent it from selecting some types of implementations (see [Keß95]). The result is a text file containing the user program with a call to a (parallel) library routine inserted for each pattern instance, formulated in a high–level programming language (e.g., a parallel C or FORTRAN dialect).

For a *shared memory* target machine, we then simply can omit the handling of array distributions and message passing in the parallel implementations, which makes the whole system much simpler. The code driver just prints the matched abstract syntax tree and emits the pattern instances as calls to library routines, the parameters in a specific format (we will work out this in section 4).

It is important to note that the library code already contains the mapping of (implicitly formulated) parallel tasks to processors; thus we need not to consider any scheduling problems afterwards.

We however want to state the following key difference between (future) *distributed-memory* back-ends of PARAMAT and current scalable, data-distribution-independent parallel libraries for distributed-memory machines:

As PARAMAT performs a global optimization of array distributions at compile time, the final configuration of array distributions will be fixed already at compile time, and the parallel code induced by these distributions will be generated *after* that.

It follows that this code generation corresponds to a partial evaluation of the implementation specification at compile time. Thus, it would be less advisable to have a large library of parallel routines that have been made data distribution independent by accessing formal parameters for array distributions, with the final program being only managed by the linker. Instead, we should provide a parameterized (data distribution independent) *specification* of such parallel routines that can be read and partially evaluated by the compiler, but the compiler's output is a set of routines that are data distribution *dependent.*

The parameters of these output routines will thus contain no information on array distributions any more — thus, the format of parameters can be the same as that for shared memory routines. That is why we focus on the shared memory back-end in the sequel.

3.2 Parameter format

This subsection describes the general interface for passing vector and matrix parameters to PARAMAT back-end routines R. We distinguish between two cases, mainly depending on the data structure used to represent multidimensional arrays:

- the *short format* is used if for all vector or matrix operands accessing some array A, passed to that routine, the following holds:

  Array A of dimensionality $d > 1$ is implemented as a 1D array of pointers to consecutively stored subarrays of dimensionality $d-1$, as suggested e.g. in [PTVF92] (thus the extents of the array axes need not to be known for addressing), and the origin of each vector or matrix operand accessing A is equal to the origin of A, and the array dimensions accessed by the vector or matrix have unit access strides.

- the *general format* is used otherwise.

3.2.1 General format

To pass a vector v to a routine in general format, we must specify a parameter sequence `A, inc1, negv, absv`, where `A` is a pointer to the first element of the (sub-)array A that is addressed by v, `inc1` denotes the increment for v's accesses to A (which may also be negative!), `negv` is a flag that indicates whether the elements of A accessed by v should be negated before being processed further, and `absv` does the same for taking the absolute value. The extent n of v (i.e., the problem size) is passed once for all operands of a routine.

For a matrix m in general format, the parameter sequence is `A, inc1, inc2, TRANSPm, negm, absm`, where `A`, `inc1`, `negm` and `absm` are defined similar to the vector case above, and `TRANSPm` indicates whether the matrix access m to A is to be transposed or not, before being processed further. We must specify `TRANSPm` to keep the order of extents n_1, n_2 of m consistent for all operands of the routine.

In this case, the caller must know the array axis extents of A to specify the increments properly.

Example: The matrix access `-A[3][2:m][2][1:r][1]` to an array `A` with declared or allocated extents `A[k][m][n][r][s]` is passed as

```
(..., A[1+s*(1+r*(2+n*(2+m*3)))], n*r*s, s, 0, 1, 0, ...)
```

If inversions of vector or matrix operands should be required by the corresponding slot of the pattern instance, these are computed separately before the call, using a temporary variable.

3.2.2 Short format

The short format requires no specification of increments. A transposition flag is necessary, but, if applicable, computation of negation or absolute value of a vector/matrix is done separately before the call, as already done for inversions. Thus, parameter passing for a vector v involves only one parameter (the pointer to the first array element to be accessed), and for a matrix m only two parameters (the pointer and the `TRANSPm` flag).

The short format tries to avoid long library routines and eliminates some overhead involved in extensive parameter passing of the general format. The short format is justified since this "normal" case appears to occur quite frequently in practice.

3.2.3 Other parameters

Often, an initializing scalar, an initializing vector or even an initializing matrix is required by an operation, e.g. at matrix vector multiplication (MV), $y := Ax + y_0$, there is an initializer y_0 that may often be zero, but can also be an arbitrary vector access to some array (e.g., y). Accordingly, for Matrix–Matrix–Multiplication, the result matrix may be initialized by a scalar or by another matrix.

A PARAMAT routine that involves such an initializing operand, must be told whether it is a scalar `s`, a vector `v` (if applicable), or a matrix `m` (if applicable). Since the parameters are typed, we pass both possibilities, e.g. for MV (short format):

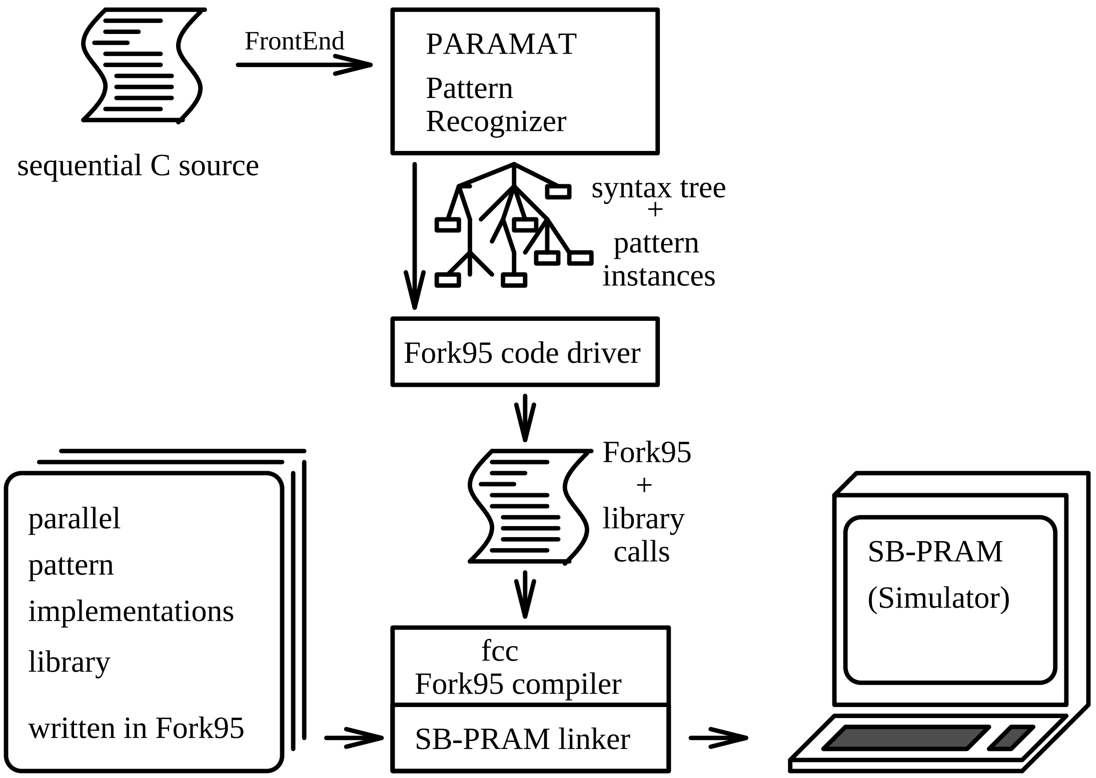


Figure 1: The overall structure of the shared-memory back-end for PARAMAT.

```
_MV( ..., s, v, ... )
```

The `_MV` routine initializes by `v` if `v` is not the null pointer; otherwise, it initializes by `s`.

4 A shared-memory back-end for PARAMAT

As a preparing step towards a future distributed–memory back–end for PARAMAT, we have built a shared-memory code driver and a library of scalable shared-memory implementations of the most important PARAMAT patterns.

We use this rudimentary back-end to demonstrate the functionality of the output produced by the pattern recognizer, and as a simple means for verification of the parallelized program, without having to build the whole system, especially, without having to care about array distributions.

As target machine, we use the SB-PRAM, a massively parallel shared memory multiprocessor with uniform memory access time. For simplicity of implementation, we have written the routines in a high-level language, Fork95 [HSS92, KS95] that we developed for the SB-PRAM in an other project.

The overall structure of the shared-memory back-end is given in Figure 1.

4.1 The target machine: SB-PRAM

The SB-PRAM [AKP90] is a lock-step-synchronous, massively parallel MIMD multiprocessor currently being built at Saarbrücken University, with up to 4096 RISC-style processing elements and with a physically shared memory of up to 2GByte with uniform memory access time. Because of its architectural properties, the SB-PRAM is particularly suitable for the implementation of irregular numerical computations, non-numerical algorithms, and database applications. Since the SB-PRAM hardware is not yet available (a 128-PE-prototype is to be expected for 1995), we use a simulator that allows to measure exact program execution times.

4.2 Library language: Fork95

Fork95 is a redesign of the PRAM language FORK. Fork95 is based on ANSI C and offers additional constructs to create parallel processes, hierarchically dividing processor groups into subgroups, managing shared and private address subspaces. Fork95 makes the assembler-level synchronity of the underlying hardware transparent to the programmer. It further enables direct access to the hardware-supplied multiprefix operations (integer only) that can be executed within 2 clock cycles.

The `start(`k`)` statement starts k processors simultaneously, with processor IDs called `$` numbered successively from 0 to $k-1$; if the expression (k) is omitted, then all `__STARTED_PROCS__` available processors are started. The processors started form one single processor group at the beginning, and execute synchronously (`sync` functions only) unless they enter the asynchronous mode via a `farm` construct. Within the `farm` body, synchronization is suspended; at the end of a `farm` environment, the current processor groups are synchronized explicitly. Throughout the whole program, the invariant is maintained that all processors in the same group operate synchronously, and all processors in the same group have a common shared address subspace.

The SB-PRAM offers hardware features that efficiently allow the user to partition the whole shared memory into private address subspaces (one for each processor) and a shared address subspace. Accordingly, variables are classified as either private (`pr`, this is the default) or shared (`sh`), where "shared" always relates to the processor group that defined that variable. An expression is private if at least one private subexpression may occur in it; worst-case assumptions (private) have to be made e.g. at dereferencing a pointer variable, as it is not statically known whether that variable will point to a shared or a private location at run time. Shared `if` or loop conditions do not affect the synchronity, as the branch taken is the same for all processors executing it. At an `if` statement, a (potentially) private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to true form the first child group and execute the `then` part while the other processors execute the `else` part at the same time. After that, the two groups are merged again by explicit synchronization of all processors of the parent group. The available shared stack of the parent group must be subdivided among the new child groups before the splitting, and joined afterwards. Similar subgroup construction is required also at loops with private exit condition. Thus, the processor groups form a tree–like hierarchy with the starting group at the root and with the current groups at the leaves at any time of the program.

To keep everything consistent, the compiler builds shared and private group frames at each group–forming statement. In order to avoid the overhead involved in this book–keeping, the user should do private computations as far as possible within asynchronous mode (inside a `farm` body or within `async` functions).

4.3 Example

4.3.1 Source and Pattern recognizer output

The following C code of a Conjugate Gradient solver has been derived from a FORTRAN77 source program that was supplied by Prof. A.K. Louis from the Numerics department of Saarbrücken University.

Beyond the (straightforward) translation from FORTRAN77 to C, we have already applied procedure inlining and constant propagation, two pre-transformations that are required by the pattern recognizer. Furthermore, we have omitted an outer while loop that was realized by a `GOTO` construct, the initializations and the I/O statements (because these are not yet included in the pattern recognizer).

```
main() {
   double a[100][100], x[100], b[100];
   int m, n, i, j, ire, im;
   double r[100], y[100], d[100], s, rm, alpha, beta;

   for (i = 0; i<m; i++) {
     y[i] = 0.0;
     for (j = 0; j<n; j++)
       y[i] = y[i] + a[i][j] * x[j];
   }
   for (i = 0; i<m; i++)
     y[i] = y[i] - b[i];
   for (i = 0; i<n; i++) {
     r[i] = 0.0;
     for (j = 0; j<m; j++)
       r[i] = r[i] + a[j][i] * y[j];
   }
   for (i = 0; i<n; i++)
     d[i] = - r[i];
   rm = 0.0;
   for (i = 0; i<n; i++)
     rm = rm + r[i] * r[i];
   for (im = 1; im <= ire; im++) {
     for (i = 0; i<m; i++) {
       y[i] = 0.0;
       for (j = 0; j<n; j++)
         y[i] = y[i] + a[i][j] * d[j];
     }
     s = 0.0;
     for (i = 0; i<m; i++)
       s = s + y[i] * y[i];
     alpha = rm / s;
     for (i = 0; i<n; i++)
     x[i] = x[i] + alpha*d[i];
     for (i = 0; i<m; i++) {
       y[i] = 0.0;
     for (j = 0; j<n; j++)
       y[i] = y[i] + a[i][j] * x[j];
     }
     for (i = 0; i<m; i++)
```

```
      y[i] = y[i] - b[i];
    for (i = 0; i<n; i++) {
      r[i] = 0.0;
      for (j = 0; j<m; j++)
        r[i] = r[i] + a[j][i] * y[j];
    }
    beta = 1.0 / rm;
    rm = 0.0;
    for (i = 0; i<n; i++)
      rm = rm + r[i] * r[i];
    beta = rm * beta;
    for (i = 0; i<n; i++)
      d[i] = -r[i] + beta * d[i];
  }
}
```

After the pattern recognizer has been run on this source program, the shared memory code driver produces the following output:

```
/* THIS FILE WAS AUTOMATICALLY GENERATED.  DO NOT EDIT IT! */

#include "fork.h"
#include "/home/pram/include/syscall.h"
#include "bpl.h"

sh float a[100][100];
sh float x[100];
sh float b[100];
 int m;
 int n;
 int i;
 int j;
 int ire;
 int im;
sh float r[100];
sh float y[100];
sh float d[100];
sh float s;
sh float rm;
sh float beta;

int main() {
  start
  {
 _MV (__STARTED_PROCS__,m,n,y,a,0,x,0.000000,0);
 _VAADD (__STARTED_PROCS__,m,y,b);
 _MV (__STARTED_PROCS__,n,m,r,a,TRANSP,y,0.000000,0);
 _VCOPY (__STARTED_PROCS__,n,d,r);
 _VQSUM (__STARTED_PROCS__,n,&rm,r,0.000000);
  for (im=1; im<=ire; im++)
  {
 _MV (__STARTED_PROCS__,m,n,y,a,0,d,0.000000,0);
 _VQSUM (__STARTED_PROCS__,m,&s,y,0.000000);
  {sh float _temp243 = 1.0/(s)*rm;
```

```
 _VAADDSV (__STARTED_PROCS__,n,x,_temp243,d);
  }
 _MV (__STARTED_PROCS__,m,n,y,a,0,x,0.000000,0);
 _VAADD (__STARTED_PROCS__,m,y,b);
 _MV (__STARTED_PROCS__,n,m,r,a,TRANSP,y,0.000000,0);
 beta=1.0/(rm);
 _VQSUM (__STARTED_PROCS__,n,&rm,r,0.000000);
 beta=rm*beta;
 _VADDSV (__STARTED_PROCS__,n,d,beta,d,r);
  }
  }
}
```

Manually adding initializers for `m`, `n` and `ire` and code for initialization of the vectors and matrices in this program (which could also be done automatically, of course) allows to execute this program for several problem sizes.

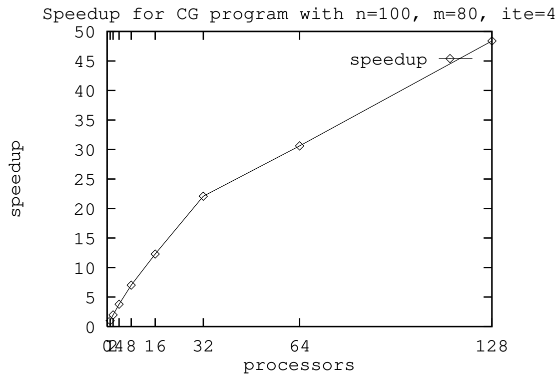
4.3.2 Library routines in Fork95

The library routines used here use the short format mentioned above. The routines are implemented straightforwardly and, of course, are still far from being optimal[2]; we show here two of them just for illustration (and as a byproduct, we would like to demonstrate how simple it is to write Fork95 programs):

```
sync void _MV (      /* short format of MV(2) matrix-vector product */
  sh int p,          /* #processors executing this call */
  sh int n,          /* length of lhs vector array access (problem size) */
  sh int m,          /* length of rhs vector array access (problem size) */
  sh vector x,       /* return value */
  sh matrix A,       /* n x m short format;  m x n if Atransp */
  sh int Atransp,    /* 1 if A transposed, 0 otherwise */
  sh vector b,       /* second operand vector, short form */
  sh float init,     /* the initialization value, if not a vector */
  sh vector initvec) /* the initialization vector, if not a scalar */
{
  pr int i, j;                                    /* private loop index */
  pr int slice = n/p + 1;                         /* local problem size */
  pr vector tmp;                                /*temp. accumulator vector */
  pr int lbi = $ * slice;                     /* lower bound of local slice */
  pr int ubi = lbi + slice;                   /* upper bound of local slice */
  farm {
    if (ubi > n)                                  /* adjust last slices */
       { ubi = n; slice = ubi-lbi; if (slice<0) slice=0; }
    tmp = (vector) malloc(slice);
    if (initvec)                                 /* preset local slice: */
```

[2]For instance, we could exploit massive parallelism better if we would also admit parallel summation in `_MV` in a tree-like manner as soon as the number of available processors exceeds the extent of the parallelized array axis. This is also the reason why all library routines have been parameterized in the number of available processors. — Furthermore, we really recommend to write a performance–critical library in assembler, also on the SB-PRAM. There are some interesting hardware properties that cannot be fully exploited by the compiler, e.g. misusing push–instructions to move data, or getting rid of some worst–case–assumptions in the compiler. In this work, we have traded performance for programming time in order to rapidly get a prototype implementation running.

Figure 2: Speedup curve for the automatically parallelized CG program. Although the library routine implementations are far from optimal and written in a high-level language, and although the local problem granularity becomes quite small for larger numbers of processors, we obtain acceptable speedups even for a modest problem size of 100 x 80.




```
      for (i=lbi; i<ubi; i++) tmp[i-lbi] = initvec[i];
    else
      for (i=0; i<slice; i++) tmp[i] = init;
  }
  if (slice) {
   if (!Atransp)
    farm
      for (j=0; j<m; j++)  /* summing loop is (still) fully sequentialized */
        for (i=0; i<slice; i++)   /* innermost: parallel saxpy computation */
          tmp[i] = tmp[i] + A[i+lbi][j] * b[j];
   else                                                 /* transposed: */
    farm for (j=0; j<m; j++)
           for (i=0; i<slice; i++)
             tmp[i] = tmp[i] + A[j][i+lbi] * b[j];
  }
  farm for (i=lbi; i<ubi; i++) x[i] = tmp[i-lbi];    /* fill result vector */
}


sync void _VAADD (  /* short format of VAADD(1) accum. vector addition */
  sh int p,         /* #processors executing this call (synchronous!) */
  sh int n,         /* length of arrays _and_ array accesses (problem size) */
  sh vector a,      /* target and additive operand vector, short form */
  sh vector v)      /* operand vector, short form.     a[] += v[]    */
{
  pr int i;                                      /* private loop index */
  farm
    for (i=$; i<n; i+=p)          /*min(n,p) processors operate in parallel */
      a[i] += v[i];                  /* each PE computes up to n/p elements */
}
```

4.3.3 Results

For the CG program with 4 iterations and the problem sizes set to `n=100` and `m=80`, we obtain the speedup ratios given in Figure 2.

The speedup curve confirms that we could exploit massive parallelism with more than 32 processors better if we would, in this case, also parallelize the summing loop in `_MV`. The optimal ratio of processors along each matrix axis depends on many factors and can only be found out empirically. Such fine–tuning needs, however, to be done only *once*, namely at the generation of the library routines. Any program using these routines will then greatly profit from their implementation. Maximal usage of these library routines also for sequential dusty deck sources can be enforced by the pattern recognition tool.

4.4 Conclusions

Although the simple back-end for the SB-PRAM is still in a preliminary state, we can already recognize some important points:

- The PARAMAT approach to automatic parallelization works fine for the SB-PRAM, even if the routines are written in a high-level programming language. Similar results for other shared memory target systems are to be expected.
- Scalability of all library routines is essential for exploiting massive parallelism also in shared memory systems.
- Fine-tuning of the library will result in increased speed-up, in particular for large numbers of processors.
- Writing library routines in a high-level language such as Fork95 is o.k. as long as pure performance is of only secondary importance.
- Following these guidelines, the classical library approach will be sufficient for shared–memory systems.

5 Towards a distributed–memory back-end

From these observations, we are able to derive guidelines for a future distributed–memory back-end of PARAMAT. The overall design of distributed-memory back-ends is given in Figure 3.

To simplify the system design being described in the sequel, we regard a given hardware environment as fixed; in particular, we regard hardware resources like number p, topology, and speed of processors, cache size and caching strategy, and memory size as constant. This corresponds to a 'dedicated' target machine. In the following considerations, we need therefore take no care any more for these hardware parameters. Nevertheless, scalability is still required since local problem granularity still depends on the problem sizes.

5.1 Synthetic performance prediction

A global optimizer determines the global distribution of all arrays occurring in the sequential source program. For optimization, this engine needs reliable run time figures to judge the quality of a specific distribution configuration. Classical bottom-up run time prediction of overall program run time given this distribution configuration is not advisable to do for larger application programs because of two reasons: First, because analytical, "low-level" performance prediction is unable to take care of typical hardware-specific effects like caching, buffering, network traffic etc.; second, because analytical performance prediction is generally too slow (e.g., the automatic array distribution engine of [DHR94] suffers mainly from slow analytical performance estimation). For this reason, we prefer "synthetic" performance prediction that assembles the global run time from table entries that have been generated off-line for each pattern, with one entry for each array distribution configuration and for each problem size magnitude (logarithmic scale). Such tables are required e.g. in [BKK94].

5.2 Automatic array distribution

A second requirement for on-line optimization of array distributions is that the parallel implementations are specified in a data-distribution-independent way. This may be done either by conditionals depending on the distribution parameters of one or several arrays, or by replication of parallel implementations, one for each possible distribution configuration. In each of these cases, it would be advisable to limit the possible distribution alternatives severely, instead of admitting arbitrary block-cyclic distributions of any block size. For vectors of length n, we allow the following distributions:

1. contiguous distribution (block size is n/p,
2. cyclic distribution (block size is 1) and
3. total replication (no distribution).

For a $m \times n$ matrix, we admit the following distributions:

1. contiguous row distribution (block size is mn/p, block shape is $(m/p) \times n$),
2. contiguous column distribution (block size is mn/p, block shape is $m \times (n/p)$),
3. cyclic row distribution (block shape is $m \times 1$),
4. cyclic column distribution (block shape is $1 \times n$),
5. contiguous quadratic blocks (block size is mn/p, block shape is $(m/\sqrt{p}) \times (n/\sqrt{p})$) and
6. total replication (no distribution, block shape is $m \times n$).

This limitation of array distribution alternatives is supported by the fact that for all our patterns [Keß94a], a locally optimal distribution for each array operand is contained in this list. We are aware of the fact that a *globally* optimal data distribution configuration may be made up of only locally suboptimal array distributions, although we believe that this scenario hardly appears in practice.

Quadratic contiguous block distributions are optimal for grid relaxation sweeps, since they minimize the surface-to-volume–ratio of the array partitions and thus the amount of data to be exchanged. In our framework, they are the only distribution scheme that distributes processors along more than one array axis. For quadratic distributions, however, we must add in this case the following constraint: The array (grid) A accessed by a matrix m must be 2–dimensional. Otherwise, imagine the following situation: Let A be three–dimensional, with axes A_1, A_2 and A_3, being distributed into quadratic blocks along, say, axes A_2 and A_3. Let m be a matrix access along the first and second axis of A. The number of processors along axis A_2 is $\sqrt{p}$, the number of processors along axis A_1 is 1 (not distributed). Thus, m has only $\sqrt{p}$ partitions, which limits parallelism unnecessarily, and, worse yet, the overall number of working processors is not constant any more for each call to the corresponding relaxation routine. Since we do not want to do everything twice, with one extra routine version for p and one for only $\sqrt{p}$ processors, we generally admit quadratic block distributions only for arrays of dimensionality equal to 2.

Problem sizes (corresponding to vector lenghts or matrix extents) need to be considered only in a specific interval $[N_{min}...N_{max}]$ of interest, e.g., from 8 to 16384. The parameter extent of that problem size axis thus contains $D = \log N_{max} - \log N_{min} + 1$ entries.

With these guidelines and with the limitation of array distribution alternatives given above, the parametrization space (and thus, the run time table size) for a pattern implementation with x vector operands, y matrix operands and z problem sizes contains $3^x \cdot 6^y \cdot D^z$ entries. For the MV matrix vector product above, we obtain an (uncompressed) table size of $54D^2$. Of course, this does not mean that we have to implement matrix vector product once for each of these configurations. Generally, several entries can be handled as a whole block, e.g., by taking array alignment relations [LC90, KLS90, KN90] into account, or ranges of problem sizes with similar run time behaviour. The run time tables can thus also be compressed according to this hierarchical parametrization structure of the parallel implementation.

5.3 Parameter format

The parameter format for distributed memory backend routines has not yet been finally determined so far. However, it seems clear that we will extend the general format for shared memory backend routines (see above) by adding array distribution parameters, e.g., one record, containing block shape and replication information, for each operand array of a routine, as e.g. done in the ScaLAPACK routines [CDWW94]. On the other hand, we can omit the first parameter indicating the available number of processors (see above).

5.4 Specification of parallel implementations

It remains the technical problem of how to code a parallel implementation in a data-distribution-independent way while maintaining explicit formulae for iteration and communication sets, and avoiding the overhead involved in evaluating complicated parametrization formulae at the target program's run time. We do this in two steps. First, PARAMAT specifies the parallel implementations in a target-machine-specific language (we favor C plus inline-assembler). This specification however allows complicated parametrization formulae or, if unavoidable, excessive replication of implementation code. Once the data distribution engine has determined a global distribution configuration for all array operands, we can derive the proper parallel implementation subroutines (comparable to those in the previous section) from that data-distribution-independent specification by partial evaluation (for reference, see [JGS93]) and dead code elimination. We obtain small and efficient message-passing–C sources that are data-distribution-dependent, and we need to extract only those routines from the specification library that are called by the matched user program. These are then compiled and linked together with the matched user program that has been produced by a suitable code driver (cf. Figure 3).

These routines extracted from the specification are also used to produce the run time tables. As this is a really tedious procedure, we plan to automatize table construction.

The realization of such a distributed-memory back-end is planned to be done in a research project starting in 1995 at Trier University, Germany.

6 Conclusion

We have presented a framework that integrates the scalable library approach and parallelizing compiler technology. It is based on fast and powerful pattern recognition in sequential source programs and considerate local algorithm replacement.

By a simplified prototype implementation using the Fork95 programming environment for the SB-PRAM, we have demonstrated the functionality of this approach for a massively parallel shared–memory target machine.

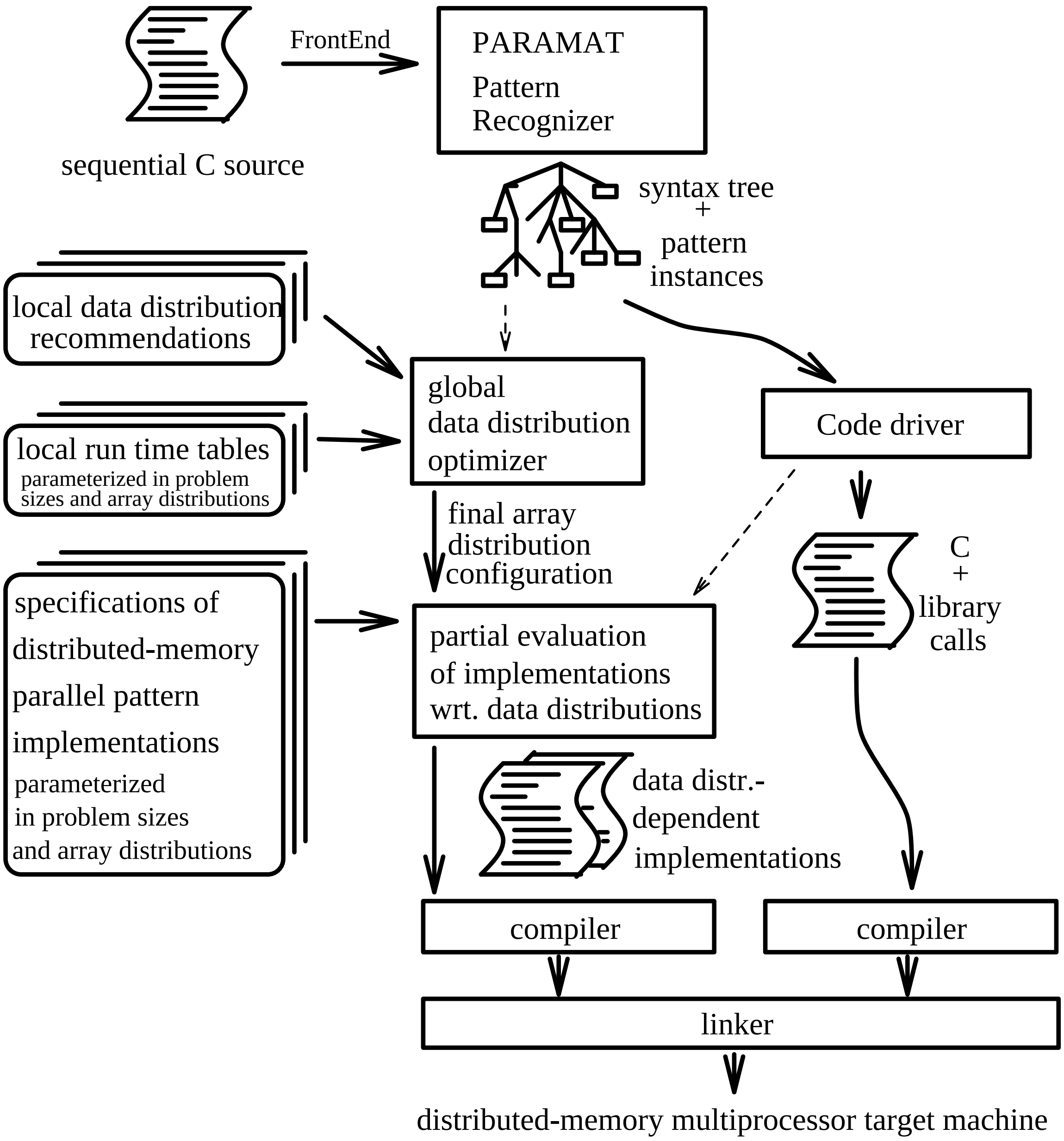


Figure 3: The overall structure of a distributed-memory back-end for PARAMAT.

We further have proposed constructive guidelines to adapt the method to distributed–memory multiprocessors, particularly by integrating an automatic array distribution engine and fast synthetic run time prediction. Extraction of data-distribution-dependent library routines is accomplished by applying partial evaluation to a data-distribution-independent specification of parallel implementations.

Acknowledgements

We thank Prof. A.K. Louis from the department of numerical mathematics at Saarbrücken University for providing the FORTRAN77 source code of the CG example program. The author further thanks the SB-PRAM system software team at Saarbrücken University for providing the hardware simulation environment and system software tools for our implementation.

Papers on PARAMAT, SB-PRAM, and Fork95

Information and papers on PARAMAT can be accessed via anonymous ftp from `alkoholix.cs.uni-sb.de` in directory `pub/PARAMAT`, and via www by `http://www-wjp.cs.uni-sb.de/paramat/index.html`. Information and papers on the SB-PRAM are accessible via ftp in `pub/sbpram`, and via www by `http://www-wjp.cs.uni-sb.de/ sbpram/index.html`. Regarding Fork95, look at ftp directory `pub/Fork95` and www page `http://www-wjp.cs.uni-sb.de/fork95/fork95.html`. The Fork95 compiler is available on `ftp.informatik.uni-trier.de` in directory `pub/users/Kessler`.

References

[AKP90] F. Abolhassan, J. Keller, and W.J. Paul. On Physical Realizations of the Theoretical PRAM Model. Technical Report 21/1990, Sonderforschungsbereich 124 VLSI Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1990.

[BKK94] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Int. Conference on Parallel Architectures and Compilation Techniques (PACT94), Montreal, Canada*, Aug. 1994.

[BL92] Ralph Butler and Ewing Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

[CDPW92] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[CDWW94] J. Choi, J.J. Dongarra, D.W. Walker, and R.C. Whaley. SCALAPACK reference manual: parallel factorization routines (LU, QR, and Cholesky) and parallel reduction routines (HRD, BRD, and TRD) (Version 1.0 beta (December 31, 1993)). Technical Report ORNL/TM-12470, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, April 1994.

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland. Birkhäuser*, pages 21–1–6, April 1994.

[DHR94] Anne Dierstein, Roman Hayer, and Thomas Rauber. Automatic parallelization for distributed memory multiprocessors. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 192–217. Verlag Vieweg, 1994.

[Fri91] Stephen S. Fried. Personal Supercomputing with the Intel i860. *BYTE*, pages 347–357, Jan. 1991.

[FSSS92] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proc. Scalable High Performance Computing Conference*, Apr. 1992.

[HSS92] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High–Level Language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[KA93] Allan D. Knies and George B. Adams. The Integrated Library Approach to Parallel Computing. In *Proc. Scalable Parallel Libraries Conference*. IEEE Computer Society Press, Oct. 1993.

[Keß94a] Christoph W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung*. PhD thesis, Universität Saarbrücken, 1994.

[Keß94b] Christoph W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, Basel, Switzerland, April 1994.

[Keß95] Christoph W. Keßler. Pattern-Driven Automatic Program Transformation and Parallelization. In *3rd EUROMICRO Workshop on Parallel and Distributed Processing, San Remo*. IEEE Computer Society Press, Jan. 1995.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[KMR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 177–186, 1990.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD machines. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 416–423, 1990.

[KP93] Christoph W. Keßler and Wolfgang J. Paul. Automatic Parallelization by Pattern Matching. In *Proc. of Second Int. Conference of the Austrian Center for Parallel Computation, Springer LNCS 734*, pages 166–181, Oct. 1993.

[Kre93] Ulrich Kremer. NP-Completeness of Dynamic Remapping. Technical Report CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Aug. 1993. see also: Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, Dec. 1993.

[KS95] Christoph W. Keßler and Helmut Seidl. Making FORK Practical. Technical Report 01/95, SFB 124 VLSI-Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken, 1995.

[LC90] Jingke Li and Marina Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In *Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, 1990.

[LC91] Jingke Li and Marina Chen. Compiling Communication–Efficient Programs for Massively Parallel Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[LT93] Oak Ridge National Laboratory and University of Tennessee. Parallel Virtual Machine Reference Manual, Version 3.2. Technical report, August 1993.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition*. Cambridge University Press, 1992.

[Thi93] Thinking Machines Corp. The CMSSL Library (V. 3.1: Optimizing Numeric Operations Across High Volume Microprocessors, 1993.