Practical Task-Oriented Parallelism for Gaussian Elimination in Distributed Memory

Gene Cooperman[1]
College of Computer Science
Northeastern University
Boston, MA 02115
gene@ccs.neu.edu

Abstract

This paper discusses a methodology for easily and efficiently parallelizing sequential algorithms in linear algebra using cost-effective networks of workstations, where the algorithm lends itself to parallelism. A particular target architecture of interest is the academic student laboratory, which typically contains many networked computers that lay idle at night. A case is made for why a task-oriented approach lends itself to the twin goals of programming ease and run-time efficiency. The approach is then described in the context of TOP-C (Task-Oriented Parallel C), an example of a system to support task-oriented parallelism. In this system, the programmer is relieved of lower level concerns such as latency, bandwidth, and message passing protocols, so as to better concentrate on higher level issues of task granularity and reduction of communication traffic. Gaussian elimination is chosen as the main example, since this algorithm is both widely used and sufficiently interesting to require non-trivial forms of parallelization for the sake of efficiency.

1 Introduction

The arrival of cheap, networked workstations and personal computers has made distributed parallelism an attractive opportunity for speeding up calculations. In particular, academic environments typically include student laboratories with such facilities often lying idle at night. Nevertheless, many of the available software tools for parallelization are either large or have a significant learning curve or both. This article describes a particularly simple form of parallelism that is easily adaptable to many tasks in linear algebra. We take Gaussian elimination as our example in this article.

[1]Partially supported by NSF Grant CCR-9509783.

While there has been a great deal of work on parallel Gaussian elimination for more specialized machines (see [20] for a good, if somewhat older survey), there has been relatively little work for general, distributed memory architectures, such as a NOW (Network of Workstations). No doubt, this is due to the relative inefficiency of typical NOW's. (It should be noted that the work on ScaLAPACK [6, 5] runs on top of both MPI and PVM and hence does apply to NOW's, although it is also meant to target more specialized, high-performance machines.) Nevertheless, the price of such a configuration is zero if one already has access for running overnight jobs remotely on laboratory machines. This article proposes an easy methodology for parallelizing linear algebra routines, taking Gaussian elimination as our main example. This will sufficiently illustrate the principles so that the reader can easily apply the same principles to other tasks in linear algebra.

The parallel tool to be used here is TOP-C [9] (Task-Oriented Parallel C). (Information on obtaining the distribution is provided at the end of this article.) Task oriented parallelism is a term that has gained usage in order to contrast it to data oriented parallelism. *Data oriented parallelism* is a style of parallelism in which opportunities for parallelism are identified typically within a loop construct in the code. The sequential code for iterating the loop is then converted to parallel code in which iterations of the loop are assigned to distinct processors. We discuss these ideas in light of a distributed architecture, where communication among processors takes place through messages, since this corresponds best to a model that takes advantage of student laboratories for "free" CPU cycles. It should be noted that data oriented parallelism is also frequently implemented on shared memory computers (formally known as a SMP, or Symmetric MultiProcessor) and on vector processors. As we shall see, the same code that runs in a distributed environment using a library from TOP-C can run on a SMP with no change in code by swapping in a second TOP-C library designed for that purpose. Further, TOP-C includes a third, sequential library that makes the use of C debugging tools, such as dbx, particularly easy.

A good example of data parallelism might be code that implements an inner product, $\sum_{i=1}^{n} u_i v_i$. If p processors are available, then up to $\lceil n/p \rceil$ of the products and the sum of those $\lceil n/p \rceil$ products can all be executed on a single processor. The $\lceil n/p \rceil$ subtotals can then be combined to find the answer.

One can arrange the partial sums to be computed according to a binary tree with p leaf nodes — each leaf node being identified with a distinct processor. The $\lceil n/p \rceil$ subtotals can then executed in $\lceil \log_2 \lceil n/p \rceil \rceil$ steps, the depth of the tree, in the obvious manner. (The binary tree is chosen to be as close to balanced as possible.)

Data oriented parallelism has been attractive because it is relatively easy for a compiler to recognize the opportunities for parallelism. With few or no hints from the program writer, the compiler can still convert many of the loops into parallel routines. This approach tends to result in a relatively fine granularity of parallelism. On vector processors and SMP processors, this type of parallelization can be very efficient. On a distributed memory architecture, a successful data parallelization must overcome the message latency. There are predictions that the overall communication bandwidth for a message (including amortized delays due to latency) will be less than the bandwidth to memory in the future. However, it is still likely to be many years before such network become economical for student laboratories.

Task oriented parallelism is a style of parallelism in which the program writer specifies opportunities for parallelism by executing multiple tasks or subroutines on distinct processors. This type of parallelism requires more effort from the program writer. However, it also allows the program writer to consider the structure of his or her algorithm to obtain further opportunities for parallelism. Further, this approach tends to yield a coarser granularity that makes it easier to overcome the message latency of a distributed memory processor.

2 TOP-C

TOP-C [9] is a system that provides a C library for easily parallelizing code. There are also related software packages written for LISP [8] and GAP [7] (Groups, Algorithms, and Programming) that apply the same methodology. TOP-C and its relatives have already been used successfully in several applications of discrete computational algebra [10, 11, 12, 13]. TOP-C functions both in a distributed and in a shared memory architecture. TOP-C also includes a shared memory library that adds the ability for processors to communicate directly through a common memory location. The use of that additional capability is not discussed here, although code designed for TOP-C under distributed memory will work without change on shared memory.

For a tutorial in programming TOP-C, it is recommended to examine [9] or to obtain the distribution, itself. This article describes only enough of the model to describe the task-oriented approach. There are many parallel tools that use a task-oriented or object-oriented approach (a selection includes [2, 3, 4, 14, 16]), and the ideas described here could be ported to many of those architectures, too.

The TOP-C model takes place in a master-slave architecture. The processor on which the jobs is begun is the master processor, and all other processors are slave processors. TOP-C runs on top of MPI [17, 19] (Message Passing Interface). A subset implementation of MPI is included with the TOP-C distribution. The programmer writes a single program, which is executed on all processors. (This is often called SPMD, or Single Program Multiple Data.) As with most implementations of MPI, this one use the UNIX utility `rsh` to spawn processes on each slave processor, although other mechanisms for starting slave processes are also possible.

The TOP-C model can best be understood through two concepts: the task and the environment. Informally, the *environment* can be thought of as a globally shared memory. We shall later see that this is only approximately true, since there is a question of when the environment is updated on each processor. The *task* is a subroutine that takes as input a *task input* (sometimes also referred to as the *task*), and returns a *task result*. It is up to the programmer to determine, based on the algorithm, what is a suitable task and environment. The programmer will then write the routines `get_task()`, `do_task()`, `get_task_result()` and `update_environment()`. These routines will implicitly define both the task and the environment.

Graphically, one can describe the TOP-C model through the following model.

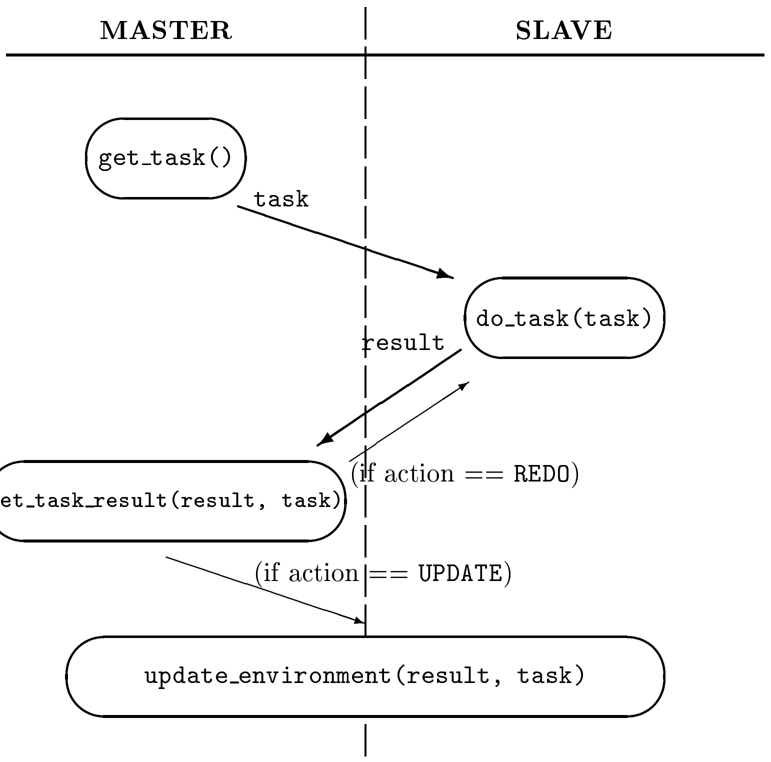

Figure 1: (This diagram appeared in [9] and is copyright by IEEE.)

If one ignores the upward arrow of the diagram, it is clear how this can be viewed as a form of trivial parallelism. Even as a tool for trivial parallelism, TOP-C can ease the programming chore, as illustrated by the following program for matrix multiplication. It should be understood that under TOP-C, the program below will be run on all processes, and the `master_slave()` routine

```
while ( NOTASK <> (task = get_task()) ) do                  [on master]
  redo:
  result = do_task( task );                                  [on slave]
  action = get_task_result( result, task );                 [on master]
  switch ( action )
    case NO_ACTION: /* do nothing */;
    case UPDATE:  update_environment(result, task); [on master & slave]
    case REDO:    goto redo;
    case CONTINUATION:  ;
```

will arrange to call the appropriate user-defined routines, according to whether the process is the master process or the slave process.

```
#define DIM 100
int mat1[DIM][DIM], mat2[DIM][DIM], mat_prod[DIM][DIM];
void *get_task() {
  static int row = -1; /* row remembered betw. calls */
  row++;
  if (row >= DIM) return NOTASK;
  return MSG(&row, sizeof(row)); }
void *do_task(void *row_ptr) {
  int i, j, row = *(int *)row_ptr;
  int result[DIM];
  for (i = 0; i < DIM; i++) {
    result[i] = 0;
    for (j = 0; j < DIM; j++ )
      result[i] += mat1[row][j] * mat2[j][i]; }
  return MSG(result, DIM*sizeof(*result)); }
void *get_task_result(int *result, void *row_ptr) {
  int i, row = *(int *)row_ptr;
  for (i = 0; i < DIM; i++)
    mat_prod[row][i] = result[i];
  return NO_ACTION; }
int main() {
  read_matrices(mat1, mat2); /* into all processors */
  master_slave(get_task, do_task, get_task_result, NULL);
  if (is_master()) print_matrix(mat_prod); }
```

In order to take advantage of non-trivial parallelism, one must bring into play a global environment, shared across processors. The environment need not be explicitly declared by the user. Instead, the user defines a routine, `update_environment()`, and any non-local data structures modified by that routine are, by definition, in the environment. The routine, `update_environment()`, is invoked on all processors whenever the user routine, `get_task_result()`, returns the action, `UPDATE`. The environment is typically initialized identically on all processors before the first call to *master_slave()*. It is the user's responsibility to insure that the environment is never modified by any routine on any processor, unless that routine was called by `update_environment()`. (More generally, a routine modifying the environment must be a descendant of `update_environment()` in the call graph.) A detailed example using the environment is contained in section 3.3.

Thus, the basic model of TOP-C is simple, and yet, as we shall see, surprisingly powerful. There are enhancements of TOP-C that are not discussed here. The most important of these is a utility, `up_to_date()`, that can be called within `get_task_result()` to check if the environment had changed between the time when the task was originally generated and the time when the result of the task was delivered. This makes possible a standard idiom by which users can define `get_task_result()`.

```
int get_task_result(void *result, void *task)
{ if ( result == NULL ) return NO_ACTION;
  if ( ! is_up_to_date() ) return REDO;
  else return UPDATE; }
```

Two other enhancements are the *continuation*, which allows an arbitrary conversation between the master and slave before a result is returned by the slave, and *raw_master_slave()*, which is useful for parallelizing sequential code in which the task is generated inside several nested loops.

3 Gaussian elimination

We first consider a simple, sequential implementation of Gaussian elimination that we wish to parallelize. Naturally, there are many sophisticated optimizations that could be applied both to the sequential and parallel versions. We omit such considerations for simplicity of exposition.

In particular, we even ignore issues of partial pivoting and numerical stability. To avoid partial pivoting, we may assume that the matrix of interest is column-wise diagonally dominant ($|a_{jj}| > \sum_{i\neq j} |a_{ij}|$), and note that Gaussian elimination preserves such a property. Such matrices are common in PDE solvers. Even so, the fact that slaves must operate in parallel may lead to effects similar to partial pivoting. If the principles of parallelization are clear, then it will also be clear to the reader how to add appropriate partial pivoting to the model, afterwards.

```
int n; /* n = matrix dimension */
float *matrix; /* matrix cast to type:  float matrix[n][n] */
```

```
int main()
{ int row;
  for (row = 0; row < n; row++)
    for (i = row + 1; i < n; i++)
      reduce_row(matrix, i, row, row); /* row already reduced */
}

void reduce_row(float *matrix, int row_to_red, int row, int pivot)
{ int j;
  float scalar = matrix[row_to_red*n+pivot] / matrix[row*n+pivot];
  float *row1 = &(matrix[row_to_red*n]);
  float *row2 = &(matrix[row*n]);
  row1[pivot] = 0.0;
  for (j = pivot+1; j < n; j++)
    row1[j] = row1[j] - scalar * row2[j];
}
```

3.1 Natural formulation

We now consider an implementation of Gaussian elimination based on TOP-C. Consider a $n \times n$ matrix. The idea for parallelization is developed in a natural manner. One would like to consider a row operation ($\bar{v} - a\bar{u}$, for scalar a and row vectors $\bar{u}$ and $\bar{v}$) as the basic task. A master would then generate such tasks for each slave, and the environment, or current status of the Gaussian elimination would be known only to the master. However, this approach does not provide sufficiently coarse granularity. A single row operation typically takes very little time, and the computation time would be dominated by the associated communication time.

3.2 Coarser granularity

So, we consider a formulation of Gaussian elimination with larger tasks (coarser granularity). We imagine that the matrix is divided into b *bands*, consisting of n/b adjacent, horizontal rows. (For simplicity of exposition, we assume that b divides n, although this is clearly not a requirement of the method.) Given p processors, we further assume that $n/b \gg 1$ and $b \gg p$. (The term, band, is used here only in its english meaning, and should not be confused with its use in banded matrices.)

This approach provides a coarser granularity. But it also requires larger messages. If one wishes to do row operations involving two bands, one must send $2bn$ numbers. While the coarser granularity may solve the problem of message latency, the communication bandwidth becomes a problem.

3.3 Lowering communication bandwidth: the environment

In order to reduce communication bandwidth, one must also make use of the TOP-C environment. The obvious candidate for the environment is the current state of the $n \times n$ matrix, and we do make that choice. We take a greedy approach, and so we define the basic task to be to reduce the band of row vectors to the maximum extent possible in the current environment. The greedy approach has clear benefits if one assumes that the communication time (including any latency) of a message dominates the time for the computation.

We take the approach of blocked Gaussian elimination. We view the matrix as a $b \times b$ matrix of $n/b \times n/b$ blocks. We define the i-th band to be *reduced* if blocks $(i, 1)$ through $(i, i - 1)$ are all zero and block (i, i) is in upper triangular form. Our goal is for all b bands to be reduced. The master process will find the next band that is not reduced and send it as the task input to a slave a be reduced. The slave process will carry out as much reduction as possible, and then return the result. If the band was further reduced by the slave, then the master process will call `update_environment()` and re-distribute that band to all processors. In order to assist in the bookkeeping, we include a global integer, `first_unred_band`, and a vector, `first_unred_col`, in the environment.

We will define the task input to be an integer that indicates the band that we will attempt to reduce for this task. We defer the definition of `get_task()` and state only that it will return an integer indicating a particular band. We assume global variables, `n` and `b` for the dimension and number of bands. For simplicity of exposition, we assume that `b` divides `n`. The routine `do_task()` can be defined as follows:

```
/* Compile with mas-slave.h and linking with TOP-C library */

int n, b;  /* n = matrix dimension; b = number of bands */
float *matrix; /* matrix cast to type:  float matrix[n][n] */
int band_size = (n+b-1)/b; /* For b | n, this is just n/b */
/* Columns 0 up to first_unred_col[b] of band b are 0 */
int first_unred_col[b];
int first_unred_band = 0; /* Done when first_unred_band = b */
int band_is_busy[b];

struct band {
  int band_no;
  int first_unred_band;
  int first_unred_col;
  float band[band_size*n];
};

void *do_task(void *band_ptr) /* 0 <= band_no <= b - 1 */
```

```
{ int band_no = *(int *)band_ptr;
  int row_start = band_no * band_size;
  int row_end = (band_no + 1) * band_size;
  int row, i;
  int pivot = max(first_unred_col[max(first_unred_band-1,0)], 0);
  static struct band result;
  /* This will always be true:  get_task() satisfied this condition */
  if (first_unred_col[band_no] <= first_unred_col[max(first_unred_band-1,0)]) {
    for (row = max(first_unred_col[band_no], 0);
         row < first_unred_band * band_size; row++)
      for (i = row_start; i < row_end; i++)
        reduce_row(matrix, i, row, row);
    /* This is non-zero block at or below first unreduced,
       on-diagonal block; Upper triangularize it in place */
    for (row = row_start; row < row_end; row++, pivot++)
      for (i = row + 1; i < row_end; i++)
        reduce_row(matrix, i, row, pivot);
    if (band_no == first_unred_band) first_unred_band++;
    first_unred_col[band_no] = first_unred_band * band_size; }
  result.band_no = band_no;
  result.first_unred_band = first_unred_band;
  result.first_unred_col = first_unred_col[band_no];
  for (i = 0; i < n * band_size; i++) /* copy band */
    result.band[i] = matrix[band_no*band_size*n + i];
  return MSG(&result, sizeof(result));
}

int get_task_result(void *res_ptr, void *band_ptr)
{  return UPDATE;
}

void update_environment(void *res_ptr, void *band_ptr)
{ struct band *result_ptr = res_ptr;
  float *mat_ptr = result_ptr->band;
  int i, band_no = *(int *)band_ptr;
  if (result_ptr->first_unred_band > first_unred_band)
    first_unred_band = result_ptr->first_unred_band;
  first_unred_col[result_ptr->band_no]
    = result_ptr->first_unred_col;
  for (i = band_no*band_size*n; i < (band_no+1)*band_size*n; i++)
    matrix[i] = *(mat_ptr++);
  band_is_busy[band_no] = 0; /* (Only master needs this) */
}
```

```
int main()
{ int i;
  n = 100;  /* set dimension */
  b = 20;   /* number of bands */
  band_size = (n+b-1)/b; /* For b | n, this is just n/b */
  for (i = 0; i < b; i++) {
    /* Columns 0 until first_unred_col[i] of band i are 0 cols and
      next block is upper triangularized; -1 means not triangularized */
    first_unred_col[i] = -1;
    band_is_busy[i] = 0; }
  matrix = malloc(n*n*sizeof(*matrix));
  master_slave(get_task, do_task, get_task_result,
       update_environment);
}
```

3.4 Load balancing

Load balancing is a typical problem in any algorithm for Gaussian elimination. In our software architecture, load balancing reduces to the control strategy used by the `get_task()` on the master process to decide which band to send out for reduction to the next available slave.

In the area of control strategy, there is room for experimentation. However, we suggest a control strategy which we have found successful. In TOP-C, it is easy to inspect the detailed load balancing, since setting a single flag causes a trace to be displayed for all messages, both to and from the master process.

We are assuming that the communication time dominates the computation time for each task. We still see an overall speedup, since there can be an overlap of communication by some processors with communication by other processors. (In section 3.5, techniques are discussed for further improving this overlap of communication and computation.) Hence, the bottleneck for Gaussian elimination tends to be the number of bands that have already been reduced and broadcast to the slave processors. Accordingly, we choose a control strategy in which `get_task()` sends out to the next available slave the first band that is not yet reduced and that is not currently being worked on by another slave.

Hence, we have the following pseudo-code for `get_task()`.

```
/* Copyright (c) 1997, Gene Cooperman; free use is granted */
void *get_task() {
  static int i;
  if (first_unred_band >= b) return NOTASK;
  for (i = first_unred_band; i < b ; i++)
    if (first_unred_col[i] < first_unred_col[first_unred_band - 1]
        && band_is_busy[i] == 0) {
```

```
    band_is_busy[i] = 1;
    return MSG(&i, sizeof(i)); } }
```

Note that this strategy provides work for additional slaves even while the first slave is working on the first band. If one recalls that `first_unred_col[i]` is initially -1 for all `i` and if one reviews the logic of `do_task()`, one sees that initially each slave will be upper triangularizing the first block in a distinct band. Such advance "pre-triangularization" of a block, B, in the lower left triangle saves half the work that will be required later when one will have upper triangularized the on-diagonal block above B and will need to "zero out" block B.

3.5 Fine tuning

The success of this methodology depends on having tasks of sufficient granularity. There are several considerations by which one can overcome problems of too fine a granularity of parallelism. First of all, one may choose to have fewer bands, each of larger band size. However, one is restricted by the requirement that the number of bands should be significantly larger than the number of processors, so that processors are not idle for most of the computation.

A second technique is to set up more than one slave process on each slave processor. Thus, one is better able to overlap computation and communication, since while a slave process is communicating with the master, a second slave process on the same processor may be computing at the same time.

This type of overlapping of computation and communication is sometimes known as *latency hiding*, since the processors remain occupied with useful computation during the communication phase. However, one should be warned that because most operating systems were designed primarily with sequential computation in mind, the operating systems may allow less than the full amount of such overlapped computation and communication. Nevertheless, there should be a tendency for this situation to improve with future operating system upgrades.

4 Comments on efficiency

See [15, Chapter 6] for an excellent introduction to practical issues of efficient, parallel matrix computations. For one of the most efficient parallel implementations of linear algebra, see the ScaLAPACK [6, 5, 21] distribution, which is part of the LAPACK [1] series. The currently most efficient implementations of Gaussian elimination do not send entire bands within a single message. As is well known, the decomposition of the matrix into blocks and the order in which the block matrix multiplications are performed allows one to lower the communication overhead and improve load balancing. Such considerations are also important in improving the cache performance, even in sequential implementations [18]. The purpose of this article is to describe a general methodology of parallelization, which can apply to novel problems in linear algebra, while yielding good (although less than optimal) performance with relatively little effort on the part of the programmer.

One of the advantages of the current methodology, as compared to more standard methods, is that dynamic load balancing is accomplished implicity. There are no barriers and no critical sections of code. Typically, no processors are starved for work, except for a short time at the end. Hence, if one processor is slow (perhaps due to external effects of a time-sharing environment), then the other processors do not usually wait for the slow processor to finish.

Further, all data transfers take place a band at a time. In architectures for which there is a significant start-up time to transfer data between processors, this can be an important consideration. However, a disadvantage of the current method is that the total amount of data transferred may be larger than other methods. This frequently happens, for example, near the beginning of the computation, when processors begin to "pre-triangularize" a band (see section 3.4) while the first slave process is still upper triangularizing the first band of the matrix. These additional bands must still be sent out and returned to the master, only to be sent out again after the first band has been upper triangularized.

It is possible to alleviate this additional communication overhead by having a slave check with the master before returning a band whose diagonal block has not been upper triangularized. If the master has received updates from other slaves in the interim, then it may be possible for the current slave to receive the update and to then make continued progress on its current band before returning from the task. TOP-C supports a `CONTINUE` action that can be used to easily implement such an optimization. If necessary, a modified algorithm that worked directly with blocks instead of bands would further alleviate this situation of starvation.

As the ratio of the matrix dimension to the number of processors grows, the total time dominates the time for the idle phase. As a practical matter Gregorio Quintana has observed in a personal communication that his own experiments with QR factorization routines using MPI seem to yield good results for band sizes between 10 and 50.

Others are welcome to experiment with this approach to linear algebra by ftp'ing the distribution from `ftp://ftp.ccs.neu.edu/pub/people/gene/top-c/`. The distribution includes its own MPI subset, so as to be self-contained. The Gaussian elimination example is included. Libraries are provided so that the same application code can be run as a single, sequential program, as a distributed memory program using MPI, or as a shared memory program using threads.

5 Acknowledgements

The author thanks Gregorio Quintana and Xiaobai Sun for their comments on this paper. The author also thanks Gregorio Quintana, Reiner Staszewski and Xiaobai Sun for valuable discussions.

References

[1] E.C. Anderson and J. Dongarra, "Performance of LAPACK: A Portable Library of Numerical Linear Algebra Routines", *Proceedings of the IEEE* **81**(8), 1993, pp. 1094–.

[2] A. Baratloo, P. Dasgupta, and Z. Kedem. "Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms", *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[3] K.M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, "Integrated Support for Task and Data Parallelism", *Intl. J. Supercomputer Applications* **8**(2), 1994, pp. 80-98.

[4] K.M. Chandy and C. Kesselman, "Compositional C++: Compositional Programming", *Proceedings of the Fourth Workshop on Parallel Programming and Compilers*, Springer Verlag.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker and R.C. Whaley, *ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers-Design Issues and Performance*, LAPACK Working Note 95, `http://www.netlib.org/lapack/lawns/lawn95.ps`

[6] J. Choi, J.J. Dongarra and R.C. Whaley, "Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines", *Scientific programming* **5**(3), Fall, 1996, pp. 173–.

[7] G. Cooperman, "GAP/MPI: Facilitating Parallelism", *Proc. of DIMACS Workshop on Groups and Computation II* **28**, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, L. Finkelstein and W.M. Kantor (eds.), AMS, Providence, RI, 1997, pp. 69–84.

[8] G. Cooperman, "STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, pp. 126–132.

[9] G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), 1996, IEEE Press, pp. 141–150.

[10] G. Cooperman, L.Finkelstein, M.Tselman and B.York, Constructing Permutation Representations for Matrix Groups, *J. Symb. Comp.*, to appear.

[11] G. Cooperman and G. Havas, "Practical parallel coset enumeration", *Proc. Workshop on High Performance Computing and Gigabit Local Area Networks*, Lecture Notes in Control and Information Sciences, Springer-Verlag, to appear.

[12] G. Cooperman, G. Hiss, K. Lux, and Jürgen Müller, "The Brauer tree of the principal 19-block of the sporadic simple Thompson group", *J. of Experimental Mathematics*, to appear.

[13] G. Cooperman and M. Tselman, "New Sequential and Parallel Algorithms for Generating High Dimension Hecke Algebras using the Condensation Technique", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, ACM Press, pp. 155–160.

[14] I. Foster and K.M. Chandy, "Fortran M: A Language for Modular Parallel Programming", *J. Parallel and Dist. Comput.* 1994 (to appear)

[15] G.H. Golub and C.F. Van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press, 1996.

[16] A.S. Grimshaw, A. Ferrari and E. West, "Mentat", in: *Parallel Programming Using C++*, G.V. Wilson and P. Lu (eds.), MIT Press, 1996, pp. 383–427.

[17] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.

[18] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The cache performance and optimizations of blocked algorithms", *Fourth International Conf. on Architectural Support for Programming Languages and Operating Systems* (April 8–11, 1991), SIGPLAN Notices **26**:4 (April, 1991), pp. 63–74.

[19] Message Passing Interface Forum (author), "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications* **8**, Number 3/4, 1994.

[20] Y. Robert, *The impact of vector and parallel architectures on the gaussian elimination algorithm*, Manchester University Press and John Wiley & Sons, 1990.

[21] ScaLAPACK Home Page,
`http://www.netlib.org/scalapack/scalapack_home.html`