Case Study in KSR Programming: Finding Outliers by the Minimum Volume Ellipsoid Method

Donna Bergmark,
Cornell Theory Center

and

Mark Crovella,
University of Rochester

28 September 1992
Theory Center Technical Report
CTC92TR107

Abstract

This case study describes the enablement of a parallel application on Cornell Theory Center's KSR1, a highly parallel machine from Kendall Square Research. The application is from statistics, and exposes some interesting facets of the KSR as well as some parallel programming tools and techniques. Statistics is a novel application area for supercomputing, at least at the Cornell Theory Center. First we describe the application itself, then the approaches to parallelizing it, and finally present some results.

Contents

1 The Problem: Finding Outliers

At first, it may seem surprising that we are trying to solve a statistics problem on the Kendall Square Research's machine (KSR1), a large parallel supercomputer. Usually such highly parallel machines are reserved for simulations of physical phenomena and other computationally intensive problems. However, there are some problems in statistics that cannot be solved by direct methods, and solution strategies similar to those used for optimization in scientific modelling and simulation are needed. One example is the use of resampling methods to provide robust estimates of certain sample statistics, such as mean and covariances.

The program we are porting deals with outlier detection[1] by calculating Mahalanobis distances (roughly, the distance of a point from the population average). Points with very large distances are presumed to be outliers. The problem with this is that the means and covariance used in calculating the distance are themselves partly determined by the outliers. The more outliers in the data, the greater this effect. If there are many outliers in the multivariate data set, then distance measures may fail to reveal any of them.

The solution is to use robust estimates for the means and covariances. For example, we might look for samples of the data which exclude outliers and use these to estimate the mean and covariance. The problem, of course, is to find the right data subsets.

2 Minimum Volume Ellipsoid Method

Minimum volume ellipsoids are one way to exclude outliers from samples used to estimate means and variances. The idea is to find mean and variance estimates that lead to a minimum volume ellipsoid containing half the data[6]. This program uses the MVE technique as described by Atkinson *et al.* [2, 3].

The algorithm used to solve this problem starts with a randomly selected set of observations, with one more observation than there are variables. Next,

[1]Detection of outliers is an old problem. Given a set of data (observations), find out which observations are "outliers" - i.e. erroneous or dubious data. For example, a data set that measures ape characteristics might have measurements for each ape such as skull size, body weight, and color of hide. If we threw two dinosaurs and five freshmen into the experiment, we would expect to find these data points to be outliers.

one calculates statistics for that set, and then calculates the distance from this set of each point in the data.[2]

Take the points with the shortest distances, and make a starting set one larger than before. Repeat this until the starting set is the entire dataset. At each step, until one gets to the very end, the outliers tend not to be very close to the set in question. Thus they have larger distances and are excluded from the earlier sets. Thus any outliers, so the theory goes, are added last.

If there are lots of outliers in the data set, a single run can miss the outliers because some outliers mask the presence of others. The magnitude of the risk depends on the "shape" of the data, i.e. the ellipsoid that contains the data. By using several random initial sets, and by normalizing distances by an estimated ellipsoid size, this risk is greatly reduced, and the method leads to a clear identification of outliers. In this application, 100 different randomly chosen starting sets are used.

More formally, given dataset Y, where Y_k^T is the k^{th} of n observations on a p-variate normal population, $m < n$ observations are used to calulate basic statistics of the data (i.e. the mean, $\bar{y}$, and the covariance matrix S). The algorithm uses these estimates to compute n Mahalanobis distances. If the initial m observations include no outliers, then outliers will give rise to some large Mahlanobis distances. These can be shown on a stalactite chart [3].

Now use those $m + 1$ observations having the smallest distances as a starting set and calculate new estimates of the mean and covariances. Since outliers will usually be further from a given set than other points, they tend to be excluded in the earlier, smaller sets. Only as m approaches n, the number of observations, will outliers be included, and "stalactites" of points with large distances will terminate.

3 The Program

In this section, we give a qualitative description of the MVE program as we received it from the author. The section after this discusses our approach to

[2]Since with the earlier, small samples, the distances will have an irregular pattern, the algorithm first calculates expected distances from simulated data; these simulated distances are used to normalize the distances obtained later on from the real data. This normalization eliminates wide variations in Mahalanobis distances calculated for small subsets of the data.

parallelizing it.

3.1 General Characteristics

The original Fortran program is modularly constructed and contains no common areas. All values are passed as parameters to subroutines. Besides the main program, there are 12 subroutines including several from the Numerical Recipes library. The program was about 600 lines long.

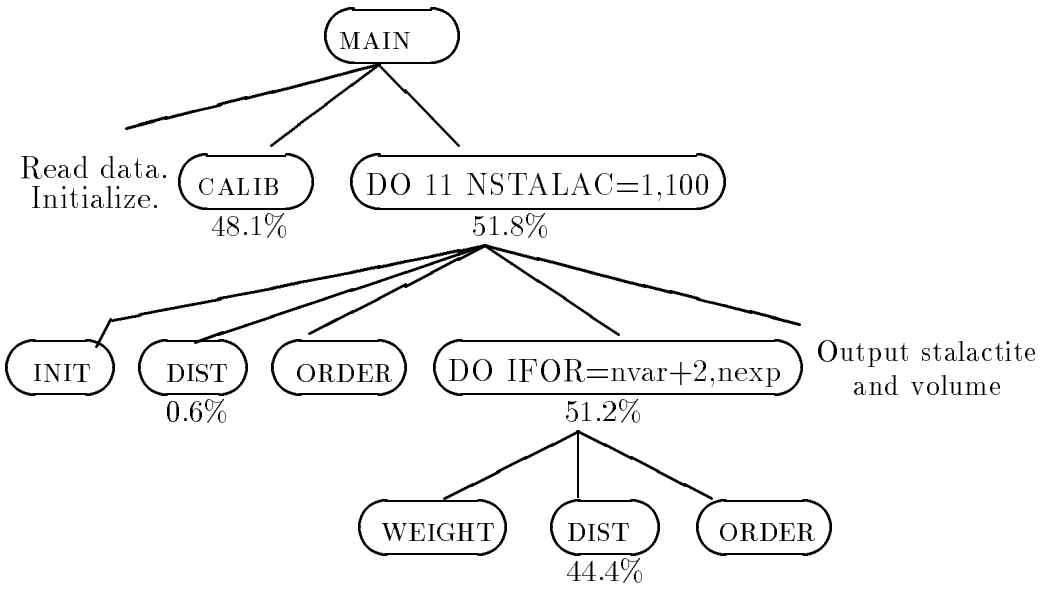
3.2 Major Data Structures

The data is read into `y`. In the current program, the user chooses which of two data sets to read in. The number of observations is stored in `nexp`, and p, the number of variables, is stored in `nvar`. The estimated volumes for various sample sizes, based on random data, are stored in the `calibvol` array. The calculated Mahalanobis distances are stored in the **dism** array. The distances are sorted by size and kept in the `ordis` array, and `label` keeps track of their original position. All four arrays are indexed by $1...nexp$.

The loop index `ifor` starts at `nvar`+2 and runs up to `nexp`; it is the current sample size. A characteristic membership function is used to represent samples: array `wtin(i)` is 1 if the i-th point is in the current subset, with $1 \leq i \leq$ `nexp`.

An abbreviated symbol table is included as Appendix A, and the organization of the program is shown in the next section.

3.3 Call Graph



The call graph shown above depicts the main parts of the program. First, the data is read in. Then CALIB is called. It estimates expected distances for variously sized samples, based on randomly generated datapoints. The running time of CALIB depends on both `nvar` and `nexp` and turns out to take 48.1% of the overall runtime to generate `nexp` expected volumes.

The main program then does 100 runs, producing one stalactite plot per run. This loop is the most time consuming portion of the program, accounting for more than half the serial runtime.

For each of the 100 runs, the program selects a random starting set; then it computes the Mahalanobis distances for all the data points; it sorts the points; and then it selects a new starting set (one larger) from the points having the least distances in the preceding set. This repeats until the starting set equals the whole set. The iterative set-expanding loop (DO IFOR) does the bulk of the work.

See Appendix B for the static call graph and loop listing of the original program.

4 Porting Preliminaries

The first job in porting a program to a new machine is to get a feeling for the compute time of the program, and also verify that the results are right. Often one also optimizes the serial program first, simply in the course of looking it over for parallelism, but we did not do this.

4.1 Execution Timing

On the sample dataset,[3] the original Fortran program runs for 11.5 minuts on the KSR. For timing, we used Forge from Applied Parallel Research and user_seconds() from KSR. The former is used to instrument the original source code, and the latter is used to provide seconds to microsecond accuracy. The original program was instrumented and run on the KSR; the resulting profile is included as Appendix C. Some of these percentages were shown on the call graph in the preceding section.

4.2 Getting the Right Results

The compiler on the Kendall Square machine at the time of this writing was at an early stage of development and often had trouble with optimization. The initial run at optimization level 2 failed to give the correct results. Running with no optimization produced the correct answers.

The standard operating procedure in this case is a binary elimination to find out which (hopefully, single) routine fails to compile correctly with optimization turned on. It took six runs to accomplish this step for our example program of 13 routines, which is close to optimal. We found that all but one routine in the program can be optimized (DIST). Unfortunately, that routine is also the most computationally intensive one. See Appendix D for details.

As of the final version of this report, we had received Version 1.0.5 of the system and this fixed the optimization problem. However, we continued to run unoptimized code for most of the numbers reported here.

[3] This dataset, taken from [4], consists of eight variables characterizing the composition of 87 containers of milk.

5 Parallelization Strategy

It is tempting to run CALIB in parallel with the DO 11 loop since each consumes approximately 50 % of the overall run time. However, this cannot be done because CALIB produces estimates used by the DO 11 loop.

Our strategy is therefore to (a) parallelize CALIB as best we can, and (b) run the iterations of the DO 11 loop in parallel. With 100 runs on 25 processors, we would expect the elapsed time to go down by 96%. The 100 iterations in parallel should take the same time as 4 iterations in serial.

5.1 The Main Loop

A run through PAT[4] turns up 84 data dependences in the DO 11 loop, all of which are accounted for by a few variables:

Variable	Dependence	How to handle
`iwt`	defined and used	replicate or privatize
`ordis`	ordered dependence	replicate or privatize
`stanvol`	output dependence	ignore*
`stanvol`	def-use dependence	replicate or privatize
`isymb`	multiple assignment	private
`label`	multiple assignment	replicate
`idum`	random seed, ordered	use a special generator

*This was the usual dependence where minimum volume depends on the order in which the volumes are examined, which here does not matter.

The second column gives the reason that the variable prevents the 100 iterations from running properly in parallel, and the third column indicates our response. In addition PAT lists a number of private variables for the loop: `volmin, fmsub, wtin[], det, nlow, ifor, i, vol, j, dism[]` and `iy`. The local arrays need to be replicated or put into a partially shared common (see below) because of language restrictions.

The most critical dependence involves `idum`, the seed for random number generation. Random number generator seeds almost always show up as data dependences, because a seed used in one loop iteration is often used in next

[4]Parallelization AssistanT Tool, from Georgia Tech, is described in [1].

to generate a new number. We solved this by writing a new generator, as described below, completely eliminating the dependences involving `idum`.

Initially some output dependences were listed because the author of the original code had left some extraneous statements after the end of the DO 11 loop. Loop generated values used in these statements would have had to be saved by the thread executing the final iteration. But in conference with the author, we discovered these statements really were not required and their deletion simplified parallelization considerably. (One additional preliminary step for parallelization often is to make the program as simple as possible.)

To parallelize the main loop, we replicated the global data areas that were both used and written into. The replication was in the final dimension, which was made to run from 1 to 100. Thus `wtin`, the set membership relation, was expanded from `wtin(1:nexp)` to `wtin(1:nexp,1:NNSIM)` where each run used `nstalac` to index into its own slice of the array: run number i would use `wtin(1:nexp,`i`)` while run number j would use `wtin(1:nexp,`j`)`. Note that this strategy keeps the `nexp` elements of the array continguous for each iteration, and thus hopefully not too many cache lines will be used.

5.2 A Parallel Random Number Generator

In serial programs, one seed is used to generate the next one. Such ordered use of a variable would be a serial bottleneck in a parallel program. Having each thread running its own copy of the generator is also not good because the independent calculations would not be using independent streams of numbers. Percus and Kalos [5] showed that you can construct a large number of independent random number generators by varying the additive constant used in linear congruential generators:

$$x_{n+1,i} \;=\; a x_{n,i} + b_i \; mod \; m$$

From a suitable collection of b_i we put together a package having 511 different generators where the different streams have a large degree of independence. The method used provides very long, reproducible sequences. To get a new random 64-bit floating point number one executes the Fortran statement,

```
r = prng_next(me)
```

where $1 \leq$ `me` ≤ 511 indentifies the calling thread, or some other index of the generator desired.

5.3 CALIB

Parallelizing CALIB turned out to be very similar to parallelizing the main DO 11 loop. This is because the CALIB routine actually has a loop structure that closely mirrors the main DO 11 loop. As a result, the approach to parallelization and elimination of loop dependencies that was worked out for the main loop was applicable to CALIB. All dependencies in the top level loop of CALIB were eliminated in the same ways as the corresponding dependencies in the main loop. The top level CALIB loop was parallelized using the same compiler directives as the main loop. As a result, the parallelization of this subroutine took very little effort.

There is a minor deficiency in the otherwise extremely expressive set of directives in KSR's parallel Fortran. This deficiency deals with reduction variables. If you have an array of reduction variables (an array of counters, say) you cannot make that a private array using partially shared commons. This was a problem for us in the parallelization of CALIB which computed the expected mean distances for 100 different-sized sets, and naturally it uses an array of sums. We had no choice but to replicate this array.

6 Results

In this section we report on various results from the parallelization effort, including the speedup achieved.

6.1 The Effect of I/O

We were interested in seeing whether writing intermediate results to a file in the middle of a parallel loop would serialize the loop, as happens on some other machines. In this sample program, the main loop writes out data for another program to use; the order of the output is not important, so long as all the output for one iteration appears together. The bulk of the output (unit 8) are the stalactites, which are graphs of zeroes and ones. Without taking special measures, parallelization causes the output to be scrambled, though

each individual record is intact. The program also sends some formatted output to standard output.

To find out whether the program was in fact being serialized by I/O, we simply commented out the entire loop I/O and compared times (see chart in Appendix E for the speedups):

Effect of I/O in Parallel Loop

# processors	1	5	10	20	30
I/O in loop	670.94	501.04	270.58	296.88	331.00
No I/O in loop	617.58	199.8	100.82	102.54	104.26

All times in seconds

Since running time actually decreases with additional processors, we conclude that I/O adds overhead but does not particularly serialize the loop.

We found that it did not much matter whether formatted I/O (i.e. standard output) went to the screen or to a file. Typical figures are shown below:

Formatted I/O

case	real time	user time	system time
to the screen	10:11.1	10:05.6	0.0
to a file	10:08.9	10:05.7	0.0

We see that writing to the console rather than to a file increases only the clock time, not the user time. The remaining runs in this work write standard output to a file.

Turning to unit 8 output, we thought we could make the program run faster by overlapping computation with I/O by using asynchronous output (which KSR does support). The first requirement is that the writes be unformatted (unit 8 initially was a formatted ascii file). The second requirement is to make sure that a variable is not being updated until asynchronous output of it has been completed. There are various ways to ensure this: one could use a separate buffer for each iteration, which would also solve our problem of keeping each iteration's output together; one could use IOCHECK or IOWAIT to see if the previous asynchronous write by this thread is complete; one could double buffer and block on I/O when the buffers are both busy, etc.

In the end, system errors prevented us from being able to replace each formatted write with a binary write, even a synchronous one. We certainly were unable to use asynchronous binary writes. In both cases, the system

gave random and confusing error messages. KSR is currently working on the problem.

Staying with formatted synchronous output for the time being, we were still faced with the problem of keeping the records together for each iteration. There are several strategies one can use:
(a) buffer up all the output until the end of the run
(b) buffer up output for each thread in a partially shared common, and output that as a single synchronous write at the end of each iteration
(c) tag each output record with the iteration number and collate the records in a separate pass
(d) use another process to collate the records on the fly, in parallel with the application
(e) write to separate files, one per thread.

Since (a) would take too much time and (d) was too hard to program and (e) was too hard to manage, we tried strategies (b) and (c). In the following, "original" is normal synchronous, formatted output to unit 8. The results are based on 1/10 and 1/2 normal run length (i.e. only 10 and 50 simulations, rather than 100). It was run on 10 processors.

Collating Strategy and Total Runtime

	10 Simulations			50 Simulations		
collating strategy	real	user	system	real	user	system
original (uncollated)	14.0	1:41	1.2	41.2	6:28	4.8
(b) buffered	14.7	1:46	1.4	44.6	6:59	4.7
(c) tagged	13.5	1:35	1.1	41.0	6:23	4.8

The results indicate that using the buffered approach costs some time, possibly because the extra memory requirements put pressure on the sub-cache. The buffered run is about 10% longer than the tagged run, but produces output that is directly usable.

Note that the I/O buffers for this problem are large. Roughly `nexp`^{2} flags (zeroes and ones) per iteration for drawing the stalactites are required. For up to 100 observations, this would take 40K bytes, more than an eighth the small cache on the machine. It would be preferable to store the data as bits. The bottom line is that I/O can seriously complicate parallelization of a program.

6.2 Private vs. Replicated Arrays

There were a number of variables in this program that would cause data dependences were the program run in parallel. The two solutions to the problem are to (1) replicate the arrays so that each run would use its own area of the array and (2) put the array into a special common and declare that common to be partially shared. (For scalar variables, the choices are similar.) When replicating, it is crucial to replicate on the final dimension, so that one iteration's values are not in the same cache line as another's.

Which produces the lowest overhead? They really both are doing the same thing – data winds up in different processors. In the first case only the i-th thread refers to the i-th column of the array so that data winds up in the cache of the processor running that thread. In the second case the system puts a private copy of the array into the cache of each processor. This might organize the data more efficiently. (Fewer data needs to be allocated in the case of several iterations being run by a single thread, because the private array for that thread can be reused.)

Making the arrays private simplifies parallelizing the program since all subscript expressions remain unchanged. That means that one could do a completely serial run since all the directives would be treated as comments. One does, however, have to build a common area and then declare it to be partially shared.

The table below shows the relative runtimes for the two alternatives. There does not seem to be much difference between them, so all things being equal it is better to use a private common since that has slightly lower runtime and is easier to code. Aligning to subpage had no affect.

Comparison of Private and Replicated Arrays

common	RUN1	RUN2	RUN3
`wtin, dsim`	replicated	private	private
`ordis, stanvol`	replicated	private	private
`label, iwt`	replicated	replicated	private
real T_1	10:49	10:47	10:46
user T_1	10:47	10:46	10:44
system T_1	6.5	6.5	6.5
real T_{26}	56.9	56.6	55.6
user T_{26}	23:43	23:34	23:09
system T_{26}	14.6	14.5	14.3

All run times are given as min:secs.tenths. T_1 is a single processor run, and T_{26} is a 26-processor run.

6.3 Speedup

Turning to parallelization results, we compare the running time on the KSR1 of various formulations of this program. First, we get some baseline figures for single processor runs:

1. unoptimized, unparallelized version. This is the baseline value for speedup results.
2. optimized (all except DIST) but not parallelized
3. single processor, unoptimized, DO 11 and CALIB parallelized, using replicated arrays.

Comparisons of Single Processor Runs

experiment number	real time (secs)	user time (secs)	system time (secs)
1	10:10.7	10:09.8	0.0
2	10:08.9	10:05.7	0.0
3	10:52.2	10:48.8	6.7

Comparing rows 1 and 2, we see the effects of optimization: only a few per cent for this code. That is because DIST, the one routine that could not be

optimized, is also computationally the most intensive one. The parallelized code in run 3 took longer because of parallel overhead. (Note the increase in system time.) However, we are encouraged that the overhead appears to be quite small. The serial runtime is roughly equivalent to the one-processor parallel runtime.

With the single processor run-times in hand, we then turned to find out how much we could speed up the program. Parallelizing CALIB and the DO 11 loop resulted initially only in a speedup of 7 no matter how many more processors we applied. The speedup steadily increased with the number of processors, but peaked at a speedup of 7 in the range of 7–8 processors. (Again, refer to Appendix E.) However, the parallelized portions account for almost all the program cycles, so more speedup should have been observed. This raised lots of questions:

- communication between processors? no - we made all iterations independent.
- synchronization? No, we use the slice strategy to chunk up the iterations ahead of time.
- serial overhead of loop parallelization? Well, high, but not enough to account for 1/7 of the program[5].
- data transfer that is not communication? That is, sending data between caches? One scenario is complete cache thrashing once you get up to 7 processors. But this is not likely, since most of the data should fit into the small cache (which is 256 Kbytes in size for data).

The answer turned out to be an environment variable, PL_ONE_SP_LONG, which controls the granularity of work assigned to threads.

6.4 Granularity

The default setting of PL_ONE_SP_LONG is 16, which meant that each thread would execute 16 iterations or more. Thus the 100 runs could not be run on

[5]The figure 1/7 comes from realizing that as one iteration begins execution, the system is setting up the thread for the next one. There is not enough calculation to overcome parallel overhead when seven iterations finish before the 8th can be set up.

25 threads, but only 100 divided by 16 which equals 6.25. Thus adding more processors could not affect the runtime! Setting this variable to 1 allocated one iteration to each thread. This got us quite different results: the "knee" in the curve moved all the way to the right to 26 processors, where the minimal runtime for this dataset is to be found (see Appendix F).

Parallelizing more loops within the DO 11 loop resulted in a slowdown. The lesson is that you really want to parallelize your code only up to a certain point. The KSR is not really a very fine grain parallel processor.

A final note is that our loop parallelization strategy is SLICE. This was based on the premise that the iterations all take about the same running time. This is not strictly true for two reasons: one is that choosing an initial starting set takes varying amounts of time depending on the random number generator and the size of the set, and because some processors on the KSR1 run at different speeds than others (for example, when your thread is running on the same processor where someone else's job is running). Therefore it might have been better to use the GRAB strategy when the system is loaded. (Measurements on an unloaded system showed SLICE to be very slightly more efficient than GRAB.)

After setting PS_ONE_SP_LONG to 1, we ultimately achieved a speedup of a little more than 12 on the 26 processors (based on the serial run time). The overall run-time dropped from 10-11 minutes for the serial version to 51 seconds for the parallelized version.

7 Enlarging the Application

The algorithm is $O(rn^2)$ where r is the number of trials and n is the number of variables times the number of observations.

It is not likely that more than 100 runs will be needed to determine the outliers. Thus, unlike most simulations, increasing the number of random trials does not necessarily improve the accuracy of the results.

However, this problem can grow in two other ways: the number of variables per observation, and the number of observations in the data set. In general it is not useful to increase the number of variables *ad infinitum*. (Atkinson observed that four variables are usually enough to characterize anything.) For large statistical samples, the number of observations will determine the run time. Also, the larger the data, the more likely it is that there

are many outliers in the data, thus making the MVE technique particularly applicable.

Running this code in parallel for large data (many observations) seems a very sensible thing to do.

8 Comments on Tools

Forge and PAT were the primary tools used to do the parallelization reported in this work. They are complementary; Forge prepares the initial timing runs that allow one to "zero in" on the parts of the program to be parallelized, and PAT assists you in parallelizing that section, especially if it is a DO loop. Using PAT enabled one of us (Mark) to parallelize two loops inside the DIST subroutine in only 90 minutes, including removing dependences and picking out the private variables and handling the one reduction. PAT would be even more useful if it could automatically replicate arrays, or generate KSR directives (currently it generates IBM and Cray syntax).

Forge was very useful in checking out intuitive feelings about variables. For example, it appeared that determinant `det` was never used by the main program; a variable trace in Forge verified that it *was* in fact used and showed exactly where. Similarly, Forge tracing revealed an array declared in the main program but used only within a single subroutine. This should have been declared as a local array. Using Forge to look at the use-def chain of variables in Fortran programs can be valuable for asynchronous I/O, where one needs to know whether an output of a variable might be pending at the point of an assignment to that variable.

A very useful thing to do in porting programs to the KSR is to fuse loops. The loop list from either PAT or Forge could be used for this. Or, ParaScope, a tool from Rice University, can be used to generate directly the new Fortran with indicated loops merged into a single loop. The KAP preprocessor occasionally proves useful in pointing out loop inversions, but we did not need this feature.

Generally speaking, the tools were (a) most useful used side by side; (b) used to double-check one's intuition about the program; and (c) useful for timing. None of these tools, except for KAP, generates KSR syntax, so source transformation was not something we did here. One of us (Bergmark) was already very familiar with the tools and could use them without effort; the

other (Crovella) got up to speed relatively quickly but had to ask about some of the ins and outs. Having an on-site tool expert is probably the best thing.

The timer situation is, as always, critical. We suffered because the KSR lacks a cumulative parallel timer, i.e. one that returns the total user CPU time used by all processes in this job so far, as well as the wall-clock time. It is, however, possible to use a partially shared COMMON to save cumulative times for each thread and then read these out when in a serial section of the program.

We cannot leave a discussion of tools without commenting on KSR's parallel debugger, **udb**. We are very positive about this debugger; it seems stable, it works with parallel programs, and it was invaluable to us on several occasions.

9 Comments on the KSR

Most of this work was done during the month of August 1992 while Crovella was visiting the Cornell Theory Center as a DARPA Parallel Processing Fellow. The machine became increasingly more stable as the month progressed. At the end of the month it had stayed up for an entire weekend, allowing many runs to be made. It also acquired a new operating system during the course of this study, when it was upgraded to OS 1.0.5. At this point, machine downs decidedly decreased in number.

Before then, the multiplicity of crashes was disconcerting. One of us (Crovella) found his password kept changing. Frequent reboots had an unexpected dividend, though; the machine was very lightly loaded for most all the runs reported here. After a reboot, there were usually not many people logged in to the system.

The KSR-1, with its cache architecture, is well-suited to statistics problems. Although in most statistical applications the input data needs to be looked at by everybody and so is inherently not data-parallel, the data is read-only. On many machines, this could result in memory contention. But on the KSR, one can expect the entire data set to be replicated into the cache of each processor working on the program. This gives the KSR an advantage over more traditional shared memory computers. In this example, each stalactite run and each operation within the run processed all the input data.

The machine seems to be easy to use, although figuring out all the environment variables can be difficult. The tiling statements make it exceptionally easy to parallelize loops. We enjoyed the exercise, found many items of interest to investigate further, and best of all, produced a parallel random number generator in the course of the work.

10 Acknowledgements

This research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State Science and Technology Foundation and members of the Corporate Research Institute. Mark Crovella was supported by a DARPA Research Assistantship in Parallel Processing.

References

[1] Bill Appelbe, Kevin Smith, and Charlie McDowell. Start/pat: A parallel-programming toolkit. *IEEE Software*, pages 29–38, July 1987.

[2] A. Atkinson. Robust estimation for outlier detection. In *Workshop on Data Analysis and Robustness*, Ascona, Switzerland, June 29–July 3 1992. Proceedings to be published one day by Birkhauser.

[3] A. Atkinson and H.-M. Mulira. The stalactite plot for the detection of multivariate outliers. *Statistics and Computing*, 3, 1992. (in press).

[4] J. Daudin, C. Duby, and P. Trecourt. Stability of principal component analysis studied by the bootstrap method. *Statistics*, 19:241–258, 1988.

[5] O. Percus and M. Kalos. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing*, 6:477–497, 1989.

[6] P. Rousseeuw and B. van Zomeren. Unmasking multivariate outliers and leverage points. *Journal American Statistical Association*, 85, 1990.

Appendix A: Abbreviated Symbol Table

(Most variables local to subroutines have been omitted from this list)

Variable		Scope	Size	Comments
aa		Task Local		For original Ran. # Gen.
calib:calibvol	output->	Parameter	100	See calibvol
calib:idum	<-in/out->	Parameter		See idum
calib:med	<-input	Parameter		
calib:nexp	<-input	Parameter		
calib:nvar	<-input	Parameter		
calib:sumdis		Local	100	Accumulates shortest sits
calib:volmin	output->	Parameter		
calib:y		Local	1000	Random data points
calibvol		Local	100	Calibrated volume/size
det		Task Local		Unused
dism		Local	100	Distances for
dist:det	output->	Parameter		current subset
dist:dism	output->	Parameter	100	See dism
dist:fmsub	<-input	Parameter		Size of subset
dist:nexp	<-input	Parameter		See nexp
dist:nvar	<-input	Parameter		See nvar
dist:wtin	<-input	Parameter	10	Subset membership
dist:y	<-input	Parameter	1000	See y
fmsub		Task Local		Size of current starting subset
gasdev:idum		Parameter		See idum
gaussj:a	<-in/out->	Parameter	100	
gaussj:ifault	output->	Parameter		
gaussj:n	<-input	Parameter		
gaussj:pivinv		Local		
idat		Task Local		Which data set to process
idum		Task Local		"seed" for Rand. # Gen.
ifor		Task Local		Current sete size (nlow..nexp)
init:idum	<-in/out->	Parameter		Random No. seed
init:nexp	<-input	Parameter		See nexp
init:nvar	<-input	Parameter		See nvar
init:wtin	output->	Parameter	100	Subset membership
isymb		Task Local	100	0 if point is close, 1 if far
iwt		Task Local	100	

```
label                    Task Local  100     Original pos. of points
ludcmp:a    <-input      Parameter    100    See y
ludcmp:d     output->    Parameter
ludcmp:indx <-input      Parameter   10      See nvar
ludcmp:n    <-input      Parameter           See nexp
med                      Task Local          # points in 1/2 volume ellipsoid
milkin:nexp output->     Parameter           Large data set
milkin:nvar output->     Parameter              "
milkin:y    output->     Parameter   1000       "
nexp                     Task Local          Number of observations
nlow                     Task Local          One more than prev sample size
nnsim                    Local               Parameter: # simulations to run
nsim                     Task Local          Used by Ran # Gen to initialize
nstalac                  Task Local          Which run, 1:100
nvar                     Task Local          Variables/observation
order:label  <-in/out->  Parameter   100     Index of orig. pos.
order:nobs   <-input     Parameter           # of data (nexp)
order:ordt    output->   Parameter    100     Sorted distances
order:t      <-input     Parameter    100     Unsorted data
ordis                    Local        100     Ordered distances
ran2:idum                Parameter            See idum
stout:dism   <-input     Parameter    100     See dism
stout:idat   <-input     Parameter            Whether small or large data used
stout:ifor   <-input     Parameter            See ifor
stout:isymb  output->    Parameter    100     See isymb
stout:nexp   <-input     Parameter            See nexp
volmin                   Task Local           Min vol calculated so far
weight:ifor  <-input     Parameter            Subset size (m)
weight:label <-input     Parameter    100     Li=1 if i-th distance
weight:nexp              Parameter               is one of minimum ones
weight:wtin   output->   Parameter    100     New sample, memb. func.
woodin:nexp   output->   Parameter    --      The smaller dataset
woodin:nvar   output->   Parameter              "
woodin:y      output->   Parameter   1000       "
wtin                     Local        100      Subset membership function
y                        Local                  1000     The input data
```

Appendix B: Static Call Graph and Loop Listing

```
***MAIN*** 114 lines MAIN calls:
1 call(s) to read     (ask user which data set to use)
1 call(s) to woodin   (routine to input small data set)
1 call(s) to milkin   (routine to input large data set)
1 call(s) to ran2     ("prime" the random number generator)
1 call(s) to calib    (calculate estimated mean distance for sample sizes)
  2 call(s) to write
  2 call(s) to order
  2 call(s) to dist
  1 call(s) to weight
  1 call(s) to init
  1 call(s) to gasdev
    2 call(s) to ran2
1 call(s) to init     (find the initial subset of nvar+1 datapoints)
  1 call(s) to ran2   (do it randomly)
3 call(s) to order    (sort the M. distances)
3 call(s) to dist     (compute M. distances for all nexp points)
  1 call(s) to ludcmp     (LU decomposition)
  1 call(s) to write
  1 call(s) to gaussj     (Gaussian elimination)
1 call(s) to weight   (get the m+1 minimum distances)
1 call(s) to stout    (output a stalactite)
  1 call(s) to write
6 call(s) to write    (various outputs from main routine)

Loop listing for the Main Routine (* = trivially parallelizable)
--------------------------------

(lines 41-42) *       do 20 isim=1,nsim
     1 call(s) to ran2
(lines 48-99)         do 11 nstalac=1,maxstal
     4 call(s) to write
     1 call(s) to init
     2 call(s) to dist
     2 call(s) to order
     1 call(s) to weight
     1 call(s) to stout
```

```
    (lines 64-86)         do 10 ifor=nlow,nexp
         1 call(s) to weight
         1 call(s) to dist
         1 call(s) to stout
         1 call(s) to order
         1 call(s) to write
        (lines 70-71) *       do 18 i=1,nexp
        (lines 82-83) *       do 12  j=1,nexp
(lines 104-106)         do 13 i=1,nexp
(lines 110-112) *       do 14 i=nlow,nexp
 7 simple loop(s)

Loop listing for CALIB (78 lines)
---------------------------------

(lines 145-146) *       do 15,i=1,nexp
(lines 147-194)         do 11 nstalac=1,ncalib
     1 call(s) to write
     1 call(s) to init
     2 call(s) to dist
     2 call(s) to order
     1 call(s) to weight
     1 call(s) to gasdev
    (lines 151-154)         do 1 i=1,nexp
         1 call(s) to gasdev
        (lines 152-154) *       do 1 j=1,nvar
             1 call(s) to gasdev
    (lines 165-186)         do 10 ifor=nlow,nexp
         1 call(s) to weight
         1 call(s) to dist
         1 call(s) to order
        (lines 174-176)         do 17 ix=1,nexp
        (lines 182-183) *       do 12  j=1,nexp
    (lines 189-193) *       do 3 i=nlow,nexp
(lines 198-201) *       do 16 i=nlow,nexp
(lines 202-205) *       do 18 i=nlow,nexp
     1 call(s) to write
 10 simple loop(s)
```

```
Loop listing for DIST (55 lines)
--------------------------------

(lines 298-301)         do 2 i=1,nvar
    (lines 300-301) *       do 2 j=1,nvar
(lines 302-307)         do 3 ix=1,nexp
    (lines 303-306)         do 4 i=1,nvar
        (lines 305-306) *       do 4 j=1,nvar
(lines 308-311)         do 5 i=1,nvar
    (lines 310-311) *       do 5 j=1,nvar
(lines 315-318)         do 8 i=1,nvar
    (lines 316-318) *       do 8 j=1,nvar
(lines 326-334)         do 6 ix=1,nexp
    (lines 328-330)         do 7 i=1,nvar
        (lines 329-330) *       do 7 j=1,nvar
(lines 340-342)         do 1,i=1,nvar
 13 simple loop(s)
```

Appendix C: FORGE Timing Data

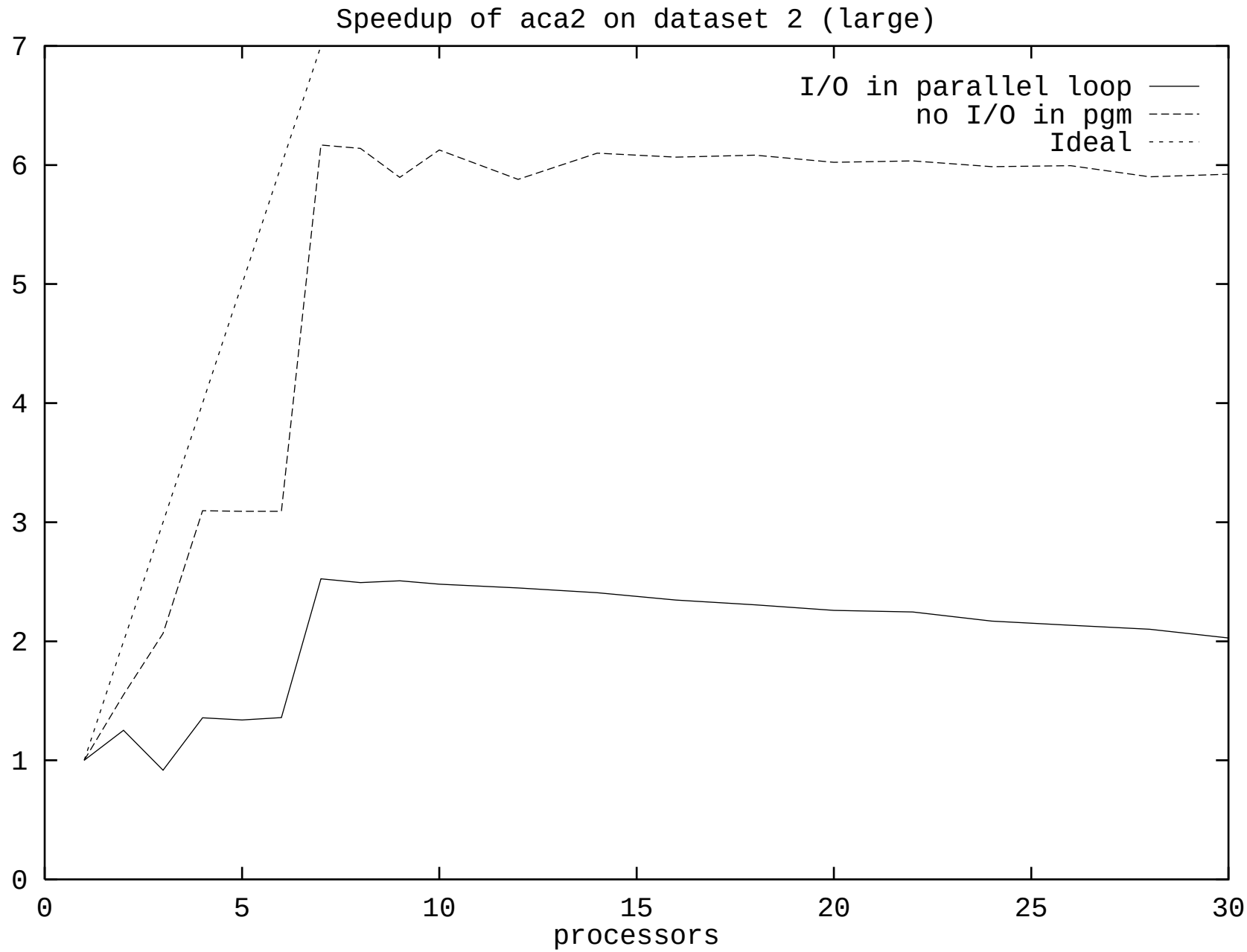
```
Viewing APR/aca2/aca.t

 >PSRTIM 20  Timing Profile for Routine aca

                                 INCLUSIVE    EXCLUSIVE DO-LOOP-LENGTH
 NEST  LOOP OR SUBPROG     %JOB:%ROUTNE %JOB:%ROUTNE     COUNT

    ACA                  100.0:100.0   0.0:  0.0           1
  1  MILKIN                 0.0:  0.0   0.0:  0.0           1
  1  DO 20 ISIM             0.0:  0.0   0.0:  0.0           1
  2   RAN2                  0.0:  0.0   0.0:  0.0          97
  1  CALIB                 48.1: 48.1   0.3:  0.3           1
  1  DO 11 NSTALAC         51.8: 51.8   0.0:  0.0           1
  2   INIT                  0.0:  0.0   0.0:  0.0         100
  2   DIST                  0.6:  0.6   0.5:  0.5         100
  2   ORDER                 0.0:  0.0   0.0:  0.0         100
  2   DO 10 IFOR           51.2: 51.2   1.3:  1.3         100
  3    WEIGHT               0.1:  0.1   0.1:  0.1        7700
  3    DIST                44.4: 44.4  39.5: 39.5        7700
  3    DO 18 I              0.2:  0.2   0.2:  0.2        7700
  3    STOUT                2.4:  2.4   2.4:  2.4        7700
  3    ORDER                2.9:  2.9   2.9:  2.9        7700
  3    DO 12 J              0.0:  0.0   0.0:  0.0        2713
  1  DO 13 I                0.0:  0.0   0.0:  0.0           1
  1  DIST                   0.0:  0.0   0.0:  0.0           1
  1  ORDER                  0.0:  0.0   0.0:  0.0           1
  1  DO 14 I                0.0:  0.0   0.0:  0.0           1
```

Appendix D: Binary Optimization

(From a pre 1.0.5 release of the OS)

ROUTINE	STEP1	STEP2	STEP3	STEP4	STEP5	STEP6
aca	none	opt	opt	opt	opt	opt
calib	none	opt	none	none	none	opt
dist	none	opt	opt	none	none	none
gasdev	none	none	none	none	opt	opt
gaussj	none	none	none	none	opt	opt
init	none	none	none	none	opt	opt
ludcmp	opt	opt	opt	opt	opt	opt
milkin	opt	opt	opt	opt	opt	opt
order	opt	opt	opt	opt	opt	opt
ran2	opt	opt	opt	opt	opt	opt
stout	opt	opt	opt	opt	opt	opt
weight	opt	opt	opt	opt	opt	opt
woodin	opt	opt	opt	opt	opt	opt
RESULTS:	OK	BAD	BAD	OK	OK	OK

Conclusion:
Binary Optimization shows that DIST is
not correctly optimized with this compiler.



Appendix E: Speedup and I/O within Parallel Loop

Appendix F: Speedup and Chunking of Parallel Loop