CMST 15(1) 57-62 (2009) DOI:10.12921/cmst.2009.15.01.57-62

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **Special Issue 2009**, 57-62

Development of Biological Warfare Sensors Using SGI High Performance Computers

Marco Lanzagorta [1], Jay Eversole [2] and Wendell Anderson [3]

[1] *ITT Corporation*

[2] *Naval Research Laboratory (Code 5612)*

[3] *Naval Research Laboratory (Code 5593)*

(Received: 31 October 2008; published online: 20 February 2009)

Astract: The Center for Computational Science (CCS) of the US Naval Research Laboratory (NRL) conducts leading edge research in High Performance Computing. CCS currently has two SGI Altix 3700s, one Altix 4700, and one SGI ICE machine. Recently the center has seen an increased interest from scientists at NRL who have been running MatLab™ on their local PC's and workstations but who need more computational power. One such application is the development of biological warfare point detection sensors where time-to-solution for a single run can take over 30 hours to complete and many runs are necessary during the development process. This paper describes the issues that were encountered in the port of this code to the SGI High Performance Computing (HPC) computers at NRL and provides a paradigm for moving other computationally intensive MatLab™ programs to HPC machines.

Key words: MatLab™, Octave, OpenMP, parallel methods, biological warfare, optical sensors

I. INTRODUCTION

Researchers at the Naval Research Laboratory often initially develop and prototype their algorithms on their local PC's and workstations using high level languages and interactive environments (such as MatLab™) that allow them to test their ideas and get results faster than by using traditional programming languages such as Fortran or C/C++. As the research evolves, the data sets grow and the algorithms increase in complexity to the point that the local available resources are insufficient to handle the increased workload and High Performance Computing resources are required. This was the situation faced by researchers in the Aerosol Optics Section of the Optical Physics Branch. A single run with a fixed set of parameters takes over hours to run data sets of 10^9 particles and larger data sets are being collected. At this rate, even the use dedicated collections of 10-20 individual desktop computers is insufficient to make adequate progress in the development and optimization of classification algorithms.

II. THE PROBLEM

The development of biological warfare (BW) point detection sensors, and to some extent chemical warfare (CW) sensors, has incorporated an approach of continuous individual aerosol particle measurement since its early beginnings around 1994 [1]. This approach is incorporated in the currently deployed DoD program of record, the Joint Biological Point Detection System (JBPDS). The optimal extraction of information from this type of sensor approach is to perform "real time" autonomous classification of each aerosol particle. NRL is participating in a current program for developing advanced BW detection capabilities sponsored by the Defense Threat Reduction Agency (DTRA) Joint Science and Technology Office (JSTO). Multiple competing mathematical approaches for achieving that sort of classification capability exist, and each approach has some parameters requiring optimisation using available experimental data. Existing data sets consist of single particle data records with between 7 and 16 measurements (feature dimensions) per particle [2, 3]. Two existing ambient background databases are roughly 10^9 and 10^{10} particles respectively in size. New additions to these large ambient background data sets are being actively acquired. Furthermore, the number of possible agent categories is large. For each of these potential threat target materials, particle size is a major defining parameter, meaning that training data sets for each target material need to span a number of different particle sizes. In essence, we

are faced with optimising potentially dozens of specific target classifiers and comparing performances for competing approaches against a continuously growing set of background clutter.

In order to meet the increasing computational requirements of this problem, the researchers were interested in using the SGI High Performance Computers (SGI Altix 3700s, SGI Altix 4700 and SGI ICE machine) available at the Naval Research Laboratory. A summary of the characteristics of these machines is given in Table 1.

Table 1. NRL SGI Computer Specifications

Machine	3700	4700	ICE
# of nodes	1	1	8
Cores per node	256	256	192
CPU Type	Itanium	Itanium	Xeon
CPU speed	1.6	1.6	3.0
L2 cache size	9 MB	18 MB	4 MB
RAM per core	8G B	4 GB	2 GB
Memory speed	667 MHZ	667 MHZ	1 333 MHZ

Historically, the algorithm development efforts both at NRL and at its collaborators at the Massachusetts Institute of Technology – Lincoln Labs (MIT-LL) were initiated in MatLab™. The cost of transcribing the algorithms developed to date into a different programming language is prohibitive within the scope of the current program. Possible options are further constrained by the requirement that code running on the SGI computers had to reproduce "exactly" the same results as on the PC's currently used.

Three options were considered for running the codes on the SGI computers – MatLab™[4], Star-P[5], and Octave [6]. MatLab™ while potentially involving the least amount of work, is no longer supported on Itaniums and thus was not considered a viable option. Star-P did support the Itaniums but had a substantial cost, especially if the code was to run on three different systems. The third option was Octave, an open source environment that will run most MatLab™ programs. It does not support the MatLab™ toolboxes but in this case the code does not use any of the toolboxes. Octave would run on all three of the SGI systems, and being Open Source, at no additional financial expenditure. Therefore, the decision was made to initially try Octave to see if it would meet the project requirements.

Since a full run of the MatLab™ code for a full run of the background data (993 files) took tens of hours, a 16 file subset of the data was selected for testing and timing. Each file contains eight features for each of approximately 900 000 particles. The training set size used in the test was 300. The running time of the test case under MatLab™ was estimated to be around 30 minutes.

III. RUNNING MatLab™ CODE UNDER OCTAVE

The first step in the porting process was to run the MatLab™ code on a PC using Octave.. Octave was installed on a PC in the CCS that already had MatLab™. Running the code produced the message:

```
error: structure has no member `GLOBAL'
```

Further analysis indicated that Octave was not able to deal with nested data structures. In order for the code to run properly, nested structures had to be changed to simple data structures. For example the nested data structure

```
RESULTS{1}.GLOBAL.RUNTIME_START
```

was changed to the non-nested data structure

```
RESULTS{1}.GLOBALRUNTIME_START
```

Once this problem was corrected the code ran successfully on the test data set.

The next step was to determine that the code produced the same answer under both environments. An initial examination of the output data showed that the answers were different. Further analysis revealed that the function shuffle that creates a permutation of the simulant data used a pseudo random number generator. Since MatLab™ and Octave generated different sequences of pseudorandom numbers different outputs were generated. To solve this problem, the random numbers generated by MatLab™ were saved into a file and the saved file was used as the source of the random numbers. After this change, the two packages gave identical results. Once identical results were being produced, the performance of Octave vs. MatLab™ was measured. Running the code under Octave took about 14 times longer than running it under MatLab™ (See Table 2), not an auspicious start to reducing the time-to-solution.

Table 2. MatLab™ vs. Octave running times

Environment	Total time secs	Bayesian time secs	% of total
MatLab™	1 694	1 483	87.5
Octave	22 334	21 788	97.6

Timers were placed into the code to measure the wall clock time required by various parts of the code. Analysis of the measured times indicated that most of the time was spent in the function `bayesian_test_02`. This function calls

two other functions: `bayesian_zero_mean_unit_stdev` to transform feature data to have zero mean and unit standard deviation and `parzen_kernel` to calculate density estimates based on a Parzen classifier with means from a training vector. The first function performs the calculations

```
for i=1:train_particles
 transformed_data(i,:)=(1./feature_stdev(:)').*
(feature_data(i,:) - feature_mean(:)');
```

while the second function calculates for each record

```
centers = train_records{i}.transformed_data;
sigma = train_records{i}.sigma;
sigma_2 = 2*sigma^2;
[ncenters nfeatures]=size(centers);
particle_table=zeros(ncenters,nfeatures);
for k=1:nfeatures
  particle_table(:,k) = particle(k);
end
distances = (particle_table - centers)';
distances_2 = sum(distances.^2);
exponential_distances = exp(-distances_2/sigma_2);
y=sum(exponential_distances)/ncenters;
```

Before porting the code to the SGI computers, several modifications were made to improve the performance of the codes. A line was added to allocate and zero the matrix `transformed_data`, so that the for-loop could be vectorized.

```
transformed_data = zeros(train_particles,train_features);
for i=1: length(train_records),
 transformed_data(i,:) =(1./feature_stdev(:)').
*(feature_data(i,:) - feature_mean(:)');
end
```

For the parzen segment of code the kernel function was in-lined and the new loop analysed to move common calculations outside the loop. The new code is

```
for i=1, length(train_records),
centers = train_records{i}.transformed_data;
sigma = train_records{i}.sigma;
sigma_2 = 2*sigma^2;
[ncenters nfeatures]=size(centers);
particle_table=zeros(ncenters,nfeatures);

for j=1:particles,
 particle = transformed_data(j,:);
 for k=1:nfeatures
   particle_table(:,k) = particle(k);
 end
distances = (particle_table - centers)';
distances_2 = sum(distances.^2);
exponential_distances = exp(-distances_2/sigma_2);
Y(j,i)=sum(exponential_distances)/ncenters;
end
```

Table 3. Modified Code Running Times

Environment	Total time secs	Bayesian time secs	% of total
MatLab™(PC)	1 694	1 483	88
MatLab™(PC) new	1 450	1 250	86
Octave(PC)	22 334	21 788	98
Octave(PC) new	19 341	18 811	97
Octave (Altix) new	15 729	15 346	98

The outputs of the modified codes were verified and the running times measured. The modifications of the code resulted in a 14% reduction in the wall clock time of the code (see Table 3). The modified code was ported to the 3700 and the results verified. Running time on the 3700 was 19% less than the PC. Approximately 98% of the time on the Altix was spent in the Bayesian calculations.

IV. MEX-C

MatLab™ is an interpretive language and thus the performance of MatLab™ codes (especially under Octave) is poor. Each of the lines in the MatLab™ code is an action on a matrix so each line operates on every element of the matrix. The execution of each assignment requires the reading of one or more matrices and the writing of a matrix, a memory intensive process.

To speed up the code, the Parzen double loop was converted to a MatLab™ Mex-C function. By combining the MatLab™ lines into a doubly nested for loop, the intermediate calculations do not need to be saved and read back but may be maintained in local registers and/or cache. The greatest difficulty in writing the function was that the C code had to handle the MatLab™ API for passing data to functions. MatLab™ passes data to its functions via the 4-tuple

```
int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]
```

where

`nlhs` is the number of output matrices
`plhs` is a structure that describes these matrices
`nrhs` is the number of input matrices
`prhs` is a structure that describes these matrices

Information about the arrays may be retrieved by the calls

```
elements = mxGetNumberOfElements(prhs[0]);
nrows = mxGetM(prhs[0]);
ncols = mxGetN(prhs[0]);
xval = mxGetPr(prhs[0]);
```

where elements is the number of elements `n` the matrix, `nrows` is the number of rows, `ncols` is the number of columns, and `xval` is a pointer to the starting address of the matrix. Output matrices can be created via

```
plhs[0] =mxCreateDoubleMatrix(nrows,ncols,mxREAL);
yval = mxGetPr(plhs[0]);
```

where `nrow` and `ncols` are the number of rows and columns of the matrix, `mxREAL` defines the data type of the matrix elements (in this case real) and `yval` is a pointer to the start of the output matrix. MatLab™ uses the Fortran column major format for storing data, not the C row major format.

While the function was 80 lines, the loop itself was only 12 lines.

```
For (j=0; j<nparticles; j++) {
 yvarx=0;
 for (i=0;i<ncenters;i++) {
  distances_2 = 0;
  for(k=0;k<nfeatures;k++){
   distances=xvar2[k*nparticles+j] - xvar1[k*ncenters+i];
   distances_2 += distances*distances;
  }
  yvarx += exp(-distances_2/sigma_2)/ncenters;
 }
 yvar[j] = yvarx;
}
```

Table 4. Code timings parzen MatLab™ vs Mex-C

Environment	Total time secs	Bayesian time secs	% of total
Octave	15 729	15 346	98
Mex-C	1 918	1 788	93

Writing the MEX-C code required a substantial initial learning curve, but once the first function was completed, writing additional functions of equal complexity required less than one-tenth of the time. Table 4 shows the improvements in the performance of the code when using the Mex-C function. The running time of the new Mex-C code was comparable to the original running time of the MatLab™ C program on the PC.

V. PARALLELIZATION

The outer loop in the parzen Mex-C function is transversed once for each particle (~1 million times) with no dependencies between particles, making this an ideal candidate for OpenMP parallelization. Octave was originally installed using the g++ C compiler. Unfortunately the versions of g++ available on the system did not support OpenMP, so Octave was reinstalled using the Intel icc compiler. The new compiler required that the < operator be defined for complex data types so that operator was added to `oct_scort.cc`. With this change the MatLab code ran successfully under the installation of Octave using the icc compiler. To create multi-threaded code, the omp include was added to the beginning of the source code file

```
#include <omp.h>
```

and the following two lines were also added before the code listed in Section IV.

```
omp_set_num_threads(ompthreads);
#pragma omp parallel for private(i, k, distances_2, distances, yvarx).
```

The first line sets the number of threads that the loop will be run in parallel, while the second line (the pragma) tells the compiler that each thread needs its own private copy of the 5 variables `i, k, distances_2, distances`, and `yvarx`. Unfortunately the OpenMP directive that reads the number of threads from an environment variable does not work within the Octave environment so the number of threads must be hard coded. The code was now run on the Altix 3700 on 16 threads (see Table 5). Parallelization of the Parzen function over 16 threads resulted in a speedup of 14 in the running time of this function and an overall speedup of about 7.5. The Bayesian calculations instead of taking 93% of the time, now took a little less than half.

Table 5. Altix 3700 1 processor vs 16 processors

Environment	Total time secs	Bayesian time secs	% of total
1 thread	1 918	1788	93
16 threads	259	126	49

Parallelization of the Parzen function over 16 threads resulted in a speedup of 14 in the running time of this function and an overall speedup of about 7.5

Further analysis of the code identified two more bottlenecks. First, the reading of the input files by MatLab™ functions was very slow so a Mex-C function was written

that first determined the size of the input file and then read its contents. This resulted in an order of magnitude speed-up in reading the raw data files. Second, the calculation of the features of files required the evaluation of the $\log_{10}$ of each element of a vector of variables. Again a Mex-C function was written to perform the logarithm and OpenMP directives inserted to run the calculation multi-threaded. Table 6 shows the improvements in performance resulting from these two changes. A 20% reduction was obtained in the running time of the program. The change in the time of the Bayesian function calculation is a result of the variance seen between individual runs of the code.

Table 6. NRL Altix 3700 16 processors

Environment	Total time secs	Bayesian time secs	% of total
Old	259	126	49
New	209	119	53

The test data was run for a varying number of threads. The results are given in Table 7.

Table 7. Benchmarks on NRL SGI Machines

Machine	3700	4700	ICE
1 thread	1 838	1898	305
2 threads	936	972	164
4 threads	533	543	98
6 threads	410	364	76
8 threads	352	299	66
12 threads	266	222	N/A
16 threads.	201	180	N/A
20 threads.	189	156	N/A

OpenMP parallelization is available only over the set of processors (cores) that share a common memory. For the 3700 and 4700 this is the entire machine. For the ICE machine OpenMP is limited to the number of cores (8) that are available on a single node. Running on only one thread the Xeon based ICE machine ran six times faster than the Itanium based Altix machines. We attribute this to a faster clock rate, a faster memory bus, and the improved architecture of the Xeon processor. For all of the SGI machines as more processors were used, the wall clock time declined as gains continue to be obtained from the use of more threads.

An examination of the kernel of the Parzen function indicates each pass through the loop requires two "additions", a multiply, and two read "memory" accesses, (where the memory access may be from cache or from memory). The large sizes of the two matrices in the kernel (`xvar1` and `xvar2`) and the storage of the data in them in column major order and not contiguously as accessed greatly increases the number of reads from memory vs cache. Thus, the kernel is memory bandwidth limited. For the test case, nearly 7 000 Gigabytes of data must be accessed. On the ICE machine where the Bayesian part of the calculation took 68 seconds, this corresponds to a 10 GB memory read rate compared to the 3.4 GB peak rate from memory (some of the data will be cached and so these rates are not contradictory).

.

VI. FULL DATA RUNS

The code was now run on the 3700 for the full data set of 993 files, The initial run failed because when MatLab™ reads a file with specified dimensions it zero pads the array if the file is too small. This was not taken into account when we wrote the Mex-C file. None of the files in our test set required zero filling, but some of the files in the full set did. After the Mex-C read function was modified to zero fill when the file size was too short, the code ran properly. The additional running time to zero fill the input arrays was insignificant compared to the running time of the total program. Production runs were also made on the 4700 and ICE machines. Based on our benchmarking runs, 32 processors were used on the 3700 and 4700 while eight cores were used on the ICE

Table 8. Full Data Time-to-solutions

Machine	Number of processors	Total time hh:mm:ss	Bayesian time hh:mm:ss	% of total
3700	32	2:36:21	1:13:52	47
4700	32	2:17:23	1:13:45	55
ICE	8	1:16:26	0:41:02	54

Even though the peak gigaflop rate of the 32 processors on the Altix machines was two times that of a node (8 cores) of the ICE machines the code ran almost two times faster on ICE than on the Altix. The improved time-to-solution for the ICE machine arises from a two times improvement of I/O hen reading data from disk and improved locality of data on ICE.

Another way of processing the data would be to divide the 993-file data set into subsets and have each instance of

the code process only a single subset. The MatLab™ code was modified so each run of the code handled only a subset of the data files. When all of the subsets have been processed the output files are merged together into a single output file. Dividing the data into 8 subsets and using 14 processors per subset, the entire data set was processed in ~37 minutes wall clock time on the 3700.

VII. CONCLUSIONS

The SGI computers at NRL provided a significant performance improvement in support of the biological sensor detection work of the Optical Sciences Division. The reduction of the time-to-solution for a single run from tens of hours to tens of minutes will permit researchers at NRL to more rapidly develop, evaluate, and field new sensor systems. While a significant effort by the CSC staff was required in installing Octave and optimizing and parallelizing the MatLab™ code, most of this was spent in the initial phases of the learning curve. In the future, researchers in Optical Science will do modifications and porting to other machines. The use of Octave rather than a commercial package meant that no additional initial or ongoing software expenses were incurred. In addition, no additional hardware was needed as the code is running on hardware already at the laboratory.

Acknowledgements

The authors are indebted to the developers and maintainers for the Octave software. We also wish to thank Dr. Nichols A. Romero of the User Productivity Enhancement and Technology Transfer (PET) Group for his guidance in optimising the MatLab™ code to run under Octave. Casey Jacobson and Jeff Wiley of NRL provide much help in explaining the working of the code. Ken Hill, the NRL SGI site rep provided much insight into the internal workings of the SGI machines that allowed us to more fully understand the timing results that we saw. Funding support for the BW algorithm effort by DTRA Joint Science and Technology Office Chemical and Biological Detection Capability Area under the Defense Technology Objective CB.50 is gratefully acknowledged.

References

[1] C. A. Primmerman, *Detection of Biological Agents.* Linc. Lab.J. **12 (1)**, 3-32 (2003).
[2] J. D. Eversole, W. K. Cary Jr., C. S. Scotto, R. Pierson, M. Spence and A. J. Campillo, *Continuous, bioaerosol monitoring using UV excitation fluorescence: outdoor test results*. Field Analytical Chemistry and Technology **15**, 205-212 (2001).
[3] V. Sivaprakasam, A. Huston, C. Scotto and J. Eversole, *Multiple UV Wavelength Excitation and Fluorescence of Bioaerosols*. Opt. Express **12**, 4457-4466 (2004).
[4] http://www.mathworks.com
[5] http://www.interactivesupercomputing.com
[6] http://www.gnu.org/software/octave