Parallel Statistical Analysis Using STAPL

Abby Malkey and Nancy M. Amato

August 7, 2014

1 Abstract

STAPL[1] (Standard Template Adaptive Parallel Library) is a parallel programing library for C++ that adopts the generic programming principles of the C++ Standard Template Library (STL)[3] created in Parasol Lab at Texas A&M University. Parallel computing involves distributing the work required to run a program over several processors to decrease the run time. Complex algorithms can be implemented using this technique, and a considerable speedup can be seen even in simple programs. Test cases are created to evaluate the performance of basic algorithms and demonstrate their programmability. Three test cases were designed and implemented, and their scalability evaluated. Using components of the STAPL library, the execution time is greatly reduced when running simple programs.

2 Introduction

As the world uses more data, larger programs processing all of that data can require more time. By distributing the work across processors, the run-time can be drastically reduced. STAPL implements parallel versions of key STL principles with the goal of simplifying the development of parallel programs while providing efficient parallel execution of the work to be done. The STAPL library provides parallel equivalents of many STL containers (pContainers) and algorithms (pAlgorithms), which are used to construct parallel programs. pViews are used to access the elements of pContainers. To demonstrate the ease-of-use of STAPL and the improved performance it provides, test cases over basic statistics algorithms were created. They include cases for finding the mean, variance, and standard deviation, computing the correlation coefficient, and identifying k-nearest-neighbors.

3 Methodology of Developing an Example

For each example, the following steps were followed to design, implement, and evalute the examples:

- Define the problem statement
- Determine input/output operations needed. For example, the number of points is taken from the command line and stored as an integer
- Outline the problem in pseudocode according to problem statement, step by step
- Determine which STAPL components can be used according to pseudocode steps
- Implement the code according to the problem statement and pseudocode
- Debug code until it runs correctly and produces desired results
- Run program and evaluate results

3.1 Code Example

Following the above method the resulting code for computing the mean would be:

```
stapl::exit_code stapl_main(int argc, char **argv)
{
  vec_type vec(boost::lexical_cast<long int>(argv[1]));
  vec_view_type vec_view(vec);
  stapl::stream<ifstream> zin;
  zin.open(argv[2]);
  stapl::serial_io(get_val_wf(zin),vec_view);
  double n = vec.size();
  double mean = stapl::accumulate(vec_view,0) / n;
  stapl::do_once( msg_val<double>( "Mean:", mean ) );
  return EXIT_SUCCESS;
}
```

4 Problem Overview

The problems that were implemented for this work are:

- Mean, Variance, Standard Deviation: calculates the average time of an algorithm that finds the mean, variance, and standard deviation of a vector of floating-point numbers.
- Correlation Coefficient: calculates the average time of an algorithm that finds the correlation coefficient of two vectors of floating-point numbers.

The equation for correlation coefficient is:

$$\frac{n\left(\sum_{i=1}^{n} x_i yi\right)-\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{\sqrt{n\left(\sum_{i=1}^{n} x_i^2\right)-\left(\sum_{i=1}^{n} x_i\right)^2} * \sqrt{n\left(\sum_{i=1}^{n} y_i^2\right)-\left(\sum_{i=1}^{n} y_i\right)^2}}$$

- K Nearest Neighbors (Single): calculates the k closest points from a vector of three dimensional points of floating-point numbers given a variable number k and a target point.

5 Experiments

5.1 Experimental Setup

All experiments were run on the Cray XE6m supercomputing system, which has a total of 576 cores. Ten core counts were used, beginning at one and doubling up to 512. Each test case was repeated 32 times and the mean and 95% confidence intervals are reported. Strong- and weak-scaling was observed for each case, with data sizes used varying for each case, depending on the problem. The results show the efficiency of STAPL and the ease with which simple algorithms can be implemented in parallel.

5.2 Evaluation Metrics

Strong- and weak-scaling are used to evaluate the results. The difference between strong and weak scalabilty refers to the amount of work the processor must perform. Strong-scaling is when a constant amount of work is processed an increasing number of processors, resulting in each having a smaller workload as the number of processors increases. Weak-scaling invloves increasing the workload and the processor count at the same rate. In an algorithm where there is no overhead in the parallel implementation (e.g., communication), strong-scaling would result in execution time decreasing linearly as core count increased, and weak-scaling would result in a flat line with little to no variation.

5.3 Mean, Variance, Standard Deviation

Overview

This test case calculates the average time of an algorithm that finds the mean, variance, and standard deviation of a vector of floating-point numbers.

Design

The general design of the code is as follows:

- create and fill vector/view (parse command line arguments for vector size and filename)

- calculate mean by summing the elements of the vector and dividing by the size of the vector
- calculate variance by squaring each element minus the mean, adding them, and dividing by the size of the vector minus 1
- calculate standard deviation by taking the square root of the variance
- output results

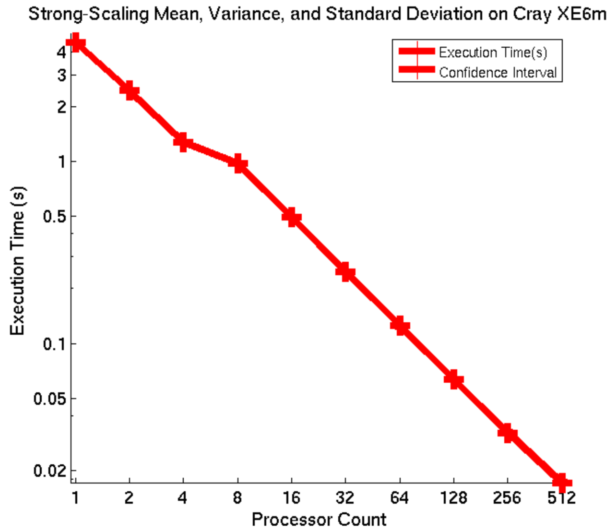
STAPL Components

The following parts of the STAPL library were used:
vector/vector view:

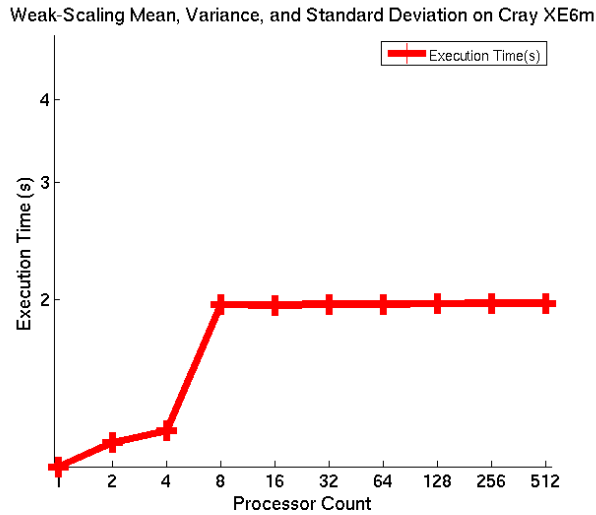
- store elements of data set to be processed
- accumulate: sums all the elements of the vector
- stream: wrapper for an STL ifstream that is used to create an input stream to read from a file
- serial_io: successively applies input operations to elements of the input view on location 0
- map_reduce: applies multiple work functions to the elements of multiple views and reduces to a single answer to be assigned to a variable
- repeat_view: repeats the mean value in the computation of variance
- do_once: performs action on only one location
- plus(): return the sum of its arguments, used for calculating variance

Results

Strong-scaling and weak-scaling trials were performed. The graphs below show the mean execution times of 32 samples with respect to the core count and 95 percent confidence intervals.



(a) Strong-scaling



(b) Weak-scaling

Analysis

The anticipated outcomes were for the times to decrease dramatically for strong-scaling and hold steady for weak-scaling. For strong-scaling, the execution time drops quickly as the processor count is increased since the work is being split among the processors. The weak-scaling results were as predicted, varying only slightly, except for a small increase when going from 4 to 8 cores. This is due to the memory bandwidth being completely utilized, as all 8 cores were placed by

the batch system onto a single die that has limited bandwidth to the memory on the compute node. The confidence intervals were small, suggesting that the mean time represents actual execution time well.

5.4 Correlation Coefficient

Overview

This test case calculates the average time of an algorithm that finds the correlation coefficient of 2 vectors of floating-point numbers.

Design

The general design of the code is as follows:

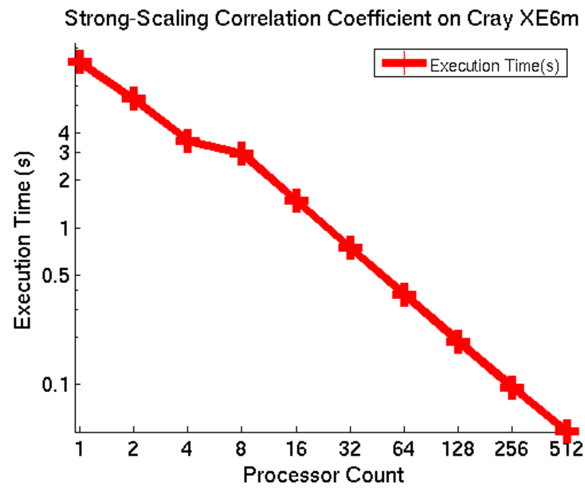
- create and fill vector/view (parse command line arguments for vector size and filename)
- calculate correlation coefficient by:
  - multiplying the size of the vectors by their inner product
  - summing each vector and multiplying the sums
  - subtracting from the inner product result
  - squaring each element in the vectors
  - taking the square root of the square of the elements and subtracting the sum of that respective vector squared
  - dividing everything into the inner product result
- output results

STAPL Components

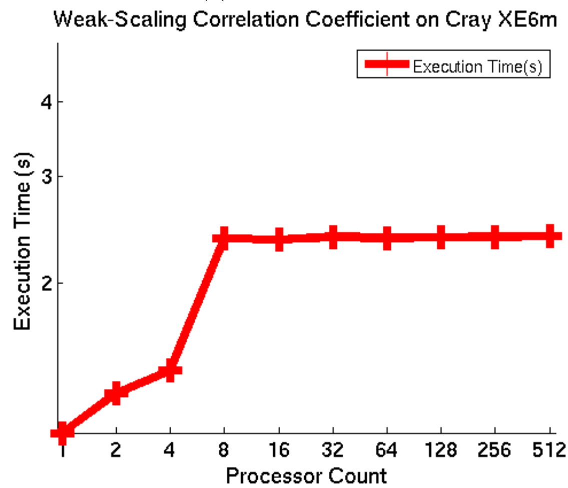
The following parts of the STAPL library were used:
vector/vector view: store elements of data set to be processed
accumulate: sums all the elements of the vectors
stream: wrapper for an STL ifstream that is used to create an input stream to read from a file
serial_io: successively applies input operations to elements of the input view on location 0
map_reduce: applies multiple work functions to the elements of multiple views and reduces to a single answer to be assigned to a variable
counter: creates a counter to record the time
do_once: performs action on only one location
inner_product: used to calculate the inner (dot) product of two vectors and adds the result to the existing result
plus(): return the sum of its arguments, used for calculating the correlation coefficent

Results

Strong-scaling and weak-scaling trials were performed. The graphs below show the mean execution times of 32 samples with respect to the core count and 95 percent confidence intervals.



(a) Strong-scaling



(b) Weak-scaling

Analysis

The anticipated outcomes were for the times to decrease dramatically for strong-scaling and hold steady for weak-scaling. For strong-scaling, the execution time drops quickly as the processor count is increased since the work is being split among the processors. The weak-scaling results were as predicted, varying only

slightly, except for a jump when going from 4 to 8 cores. This is due to the memory bandwidth being completely utilized, as all 8 cores were placed by the batch system onto a single die that has limited bandwidth to the memory on the compute node. The confidence intervals were small, suggesting that the mean time represents actual execution time well.

5.5 K Nearest Neighbors (Single)

Overview

This test case calculates the k closest points for a vector of three dimensional points of floating-point numbers and a variable number k given a target point.

Design

The general design of the code is as follows:

- create and fill vector/view (parse command line arguments for vector size, k, and target point
- create map and iterator to hold and parse results
- compare every point in the vector with the target point, calculating the distance between them
- put the results in a map with distance as the key, taking only the first k elements
- output results

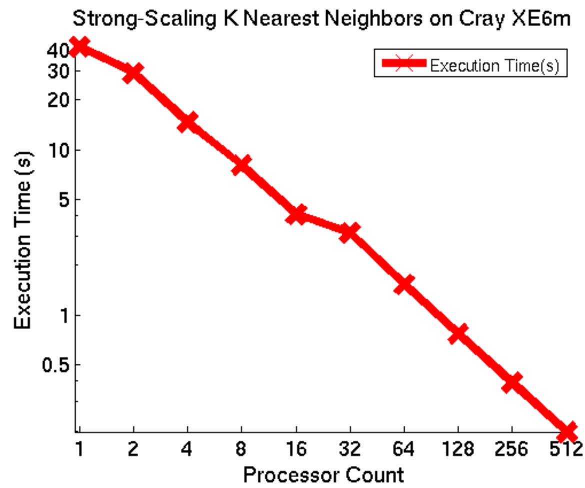
STAPL Components

The following parts of the STAPL library were used:

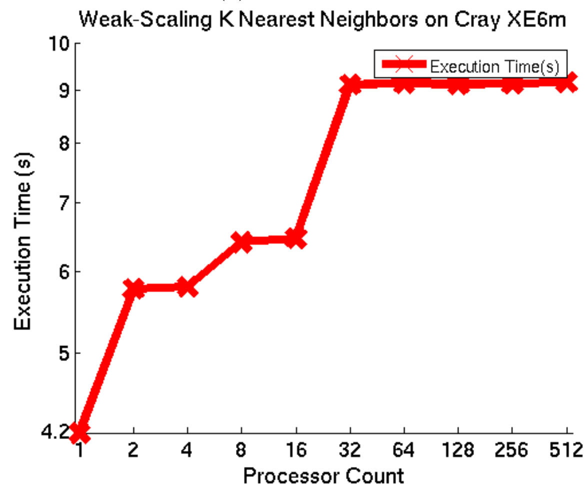
- vector/vector view: store elements of data set to be processed
- map_reduce: applies multiple work functions to the elements of multiple views and reduces to a single answer to be assigned to a variable
- do_once: performs action on only one location
- proxy specialization for point struct: allows an object to be referred to across nodes
- random_sequence: generates random doubles to fill the point vector

Results

Strong-scaling and weak-scaling trials were performed. The graphs below show the mean execution times of 32 samples with respect to the core count and 95 percent confidence intervals.



(a) Strong-scaling



(b) Weak-scaling

Analysis

The anticipated outcomes were for the times to decrease dramatically for strong-scaling and hold steady for weak-scaling. For strong-scaling, the execution time drops quickly as the processor count is increased since the work is being split among the processors. The weak-scaling results were as predicted, varying only

slightly, except for a few small jumps. When going from 4 to 8 cores, the jump is due to the memory bandwidth being completely utilized, as all 8 cores were placed by the batch system onto a single die that has limited bandwidth to the memory on the compute node. The jump between 16 and 32 cores can be attributed to the processes running on two nodes of the system instead of one. The increase from one to two cores is a bit higher than expected, but it is because the program is now running on two cores and communication is occurring, whereas no communication occurs when running on only one core. The confidence intervals were small enough to suggest that the execution time of the algorithm on a given number of processors is stable.

6 Conclusion

This work has shown that relatively simple algorithms can easily be written in STAPL. STAPL provides parallel execution which speeds up the execution time by distributing the work over a chosen number of processors, as shown by the results of the performed experiments where the execution time decreased significantly as the number of cores increased.

7 Acknowledgements

Adam Fidel and Timmie Smith provided assistance in the use of STAPL components and the design of the experimental studies.

References

[1] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedhal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger, "The STAPL Parallel Container Framework," In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), Feb 2011.

[2] Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger, "The STAPL pView," In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Oct 2010. Also, Technical Report, TR10-001, Parasol Laboratory, Department of Computer Science, Texas A&M University, Jul 2010.

[3] D. Musser, G. Derge, and A. Saini. STL Tutorial and Reference Guide, 2nd Edition. Addison-Wesley, 2001.