A Parallelization of Orchard Temperature Predicting Programs *

Elad Mazurek Munehiro Fukuda†

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011

Abstract

Frost protection is a significant concern among tree-fruit growers. In order to protect orchards from frost, temperature sensor networks have gained popularity to judge when and where to turn on wind generators and sprinklers. Currently these sensor networks only give growers a real-time alert, however such sensor networks would be more effective if they were integrated in a computation loop that uses past and current temperature data for predicting every overnight transition of orchard air temperature. Although a couple of algorithms using artificial neural network and empirically-formulated polynomials are available to the public, they need to be parallelized for useful on-the-fly prediction. To utilize a cluster of multi-core computing nodes (available through cloud services), we are developing MASS, a library for multi-agent spatial simulation, and parallelizing temperature prediction programs with MASS. This paper demonstrates the MASS library's suitability to parallelization of temperature prediction programs for on-the-fly sensor-data analysis by (1) porting the programs to MASS, (2) running them in a multi-core system, (3) feeding real-time sensor data to them, and (4) measuring their analyzing performance.

1 Introduction

Frost protection is a big concern among tree-fruit growers who suffer $840 million and $2.5 billion frost and freeze crop losses in the nation and the world respectively [8]. Currently, fruit growers use the public frost alert, their experiential knowledge, repetitive tree observation, and temperature sensor networks (if they can be afforded) for judging when and where to turn on wind generators and sprinklers to protect orchards from frost. Ideally they want to use temperature sensor networks to predict every overnight transition of orchard air temperature, which then can provide growers with accurate information to protect their crops. Temperature prediction can be practicalized using the following computation scenario: estimating the temperature of sensor-uncovered areas with temperature interpolation [3]; predicting the overnight temperature of each area with artificial neural network [9] or empirically-formulated polynomials [10]; and storing sensor data in databases for future use in prediction.

The problem is that accurate and on-the-fly prediction requires substantial computing power, more specifically only in a short frost season, (i.e., April through to June). Parallel computing with cloud services could satisfy their computation needs with as many computing nodes as possible, accompanied by cloud-common software tools such as OpenMP [7], MPI [6], and MapReduce [2]. However in reality, there are little temperature prediction programs that have been parallelized for conducting on-the-fly real-time analysis. In other words, most prediction-model designers are not specialized in parallel computing.

To facilitate parallelization of temperature prediction programs, we are currently developing MASS, a library for multi-agent spatial simulation that is based on entity-based programming [4] where an application is viewed as a series of interaction among autonomic computation entities with independent code. These entities are dynamically mapped to threads running on a cluster of multi-core machines, and their communication is automatically translated into inter-process/thread communication. Therefore, model designers utilizing MASS can focus on their application development without planning on future parallelization.

This research is to demonstrate the MASS library's suitability to parallelization of temperature prediction

*This research is being conducted with partial support from UW Provost International Grants, Faculty-led Program

†Corresponding author. Email: mfukuda@u.washington.edu, Phone: 1-425-352-3459, Fax: 1-425-352-5216

programs for on-the-fly sensor-data analysis by (1) porting the programs to MASS, (2) running them in a multi-core system, (3) feeding real-time sensor data to them, and (4) measuring their real-time analyzing performance. The rest of the paper is structured as follows: Section 2 discusses possible parallelization strategies for existing temperature-predicting programs; Section 3 sketches a system overview for on-the-fly temperature prediction with MASS; Section 4 details MASS-based parallelization of temperature prediction; Section 5 shows performance improvement with MASS parallelization; and Section 6 conclude our discussions.

2 Existing Algorithms and Technologies

2.1 Temperature Prediction and Interpolation

Temperature prediction is the key to frost protection. It takes two steps: (1) hourly-based prediction based on the past/current temperature data, and (2) temperature interpolation, a software technique to estimate temperature on each divided cell of a given land with N sampled temperature data. Due to both economic and technical reasons, the scale of a wireless sensor network installed on an orchard is generally too limited to perfectly measure the air temperature surrounding all the crops. For instance, our research collaborator uses 20 temperature sensors for his 120-acre orchard. While he wants to ultimately install 10 sensors per acre, bringing the total number of sensors to 1200, this is both economically and technically unfeasible for many farmers. Hence the motivation for temperature interpolation.

Two air-temperature prediction algorithms are available to use: one from University of California, Davis [10] and the other from University of Georgia respectively [9]. The former uses temperature T_o and dew point T_d observed at sunset as well as T_2 at two hours after the sunset; predicts the minimum temperature T_p by applying Formula 1 to the observed temperature; and plots hourly predicted temperature T_i from T_2 down to T_p using Formulae 2 and 3. (Note that these formulae are based on Fahrenheit.)

$$T_p = 0.494 \times T_0 + 0.027 \times T_d + 4.900 \quad (1)$$

$$b = \sqrt{fracT_p - T_2 n - 2} \quad (2)$$

$$T_i = T_2 + b\sqrt{i-2} \quad (3)$$

The latter uses an artificial neural network (ANN) model as shown in Formulae 4 through to 8 and predicts temperature T_{future} from one to 12 hours ahead by training ANN with seasonal parameters and weather forecast information and thereafter applying the ANN to the current temperature $T_{current}$.

$$signal_{in} = T_{current} \times weight \quad (4)$$

$$signal_a = \frac{e^{2.0 \times signal_{in}} - 1}{e^{2.0 \times signal_1} + 1} \quad (5)$$

$$signal_b = e^{-1.0 \times signal_{in} \times signal_{in}} \quad (6)$$

$$signal_c = 1 - e^{-1.0 \times signal_{in} \times signal_{in}} \quad (7)$$

$$T_{future} = \frac{1}{1 + e^{-1.0 \times (signal_a + signal_b + signal_c)}} \quad (8)$$

Two representative algorithms used to interpolate temperature are inverse distance weighting and polynomial regression [3]. Inverse distance weighting computes each cell's air temperature T, based on Formula 9 where d_i is a distance from sensor i to this cell; t_i is air temperature sampled by sensor i; and p is normally 2. The altitude of a sensor should be considered for more accuracy, in which case we estimate each cell's altitude using the same algorithm, obtain the actual altitude, and adjust T with the difference between estimated and actual altitude data. Polynomial regression solves the parameters a_0, a_1, ..., a_{10} of Formula 10 where T_i is air temperature sampled by sensor i; (X_i, Y_i) is the coordinate or the longitude and latitude pair of sensor i; and Z_i is the altitude of sensor i. The complexity of both algorithms is $O(N^2M)$, (where M is the number of cells), and is thus computation intensive.

$$T = \sum_{i=1}^{N} \left(\frac{d_i^{-p}}{\sum_{j=1}^{N} d_j^{-p}}\right) t_i \quad (9)$$

$$T_i = a_0 + a_1X_i + a_2Y_i + a_3X_i^2 + a_4Y_i^2 + a_5X_iY_i + a_6X_i^3 + a_7Y_i^3 + a_8X_i^2Y_i + a_9X_iY_i^2 + a_{10}Z_i \quad (10)$$

2.2 Parallelization Techniques

The series of temperature prediction and interpolation discussed above is so computation intensive that it needs parallelization when performing on-the-fly analysis using real-time sensor data. The parallelization of these algorithms is based on data parallelism in that an orchard is divided in sub-spaces small enough for a grower to accurately identify a space facing frost danger and to start wind generators or sprinklers there.

Typical parallelization tools available in cloud services include OpenMP [7], MPI [6], and MapReduce [2]. OpenMP is a shared-memory-based compiler-assisted parallelization to incrementally tune up these

temperature-analyzing programs but cannot be extended to distributed-memory cluster systems. Therefore, for more CPU scalability, it is a natural transition to use hybrid parallelization with OpenMP and MPI, where each MPI rank takes a collection of several orchard sub-spaces, each assigned to a different OpenMP thread within the same rank. Programmability is the main concern of this approach. For instance, we need to restrict only one thread in each process to call MPI functions while overlapping MPI communication and computation with multiple threads for better performance, which brings more complexity to such hybrid programs. MapReduce gives efficient task parallelism only if each sub-space can be thoroughly processed by an independent pair of mapper and reducer functions. Needless to say, it is not fitted for data parallelism where neighboring sub-spaces of a given orchard need to exchange their data during computation.

To address the difficulty of writing sensor-data analysis programs based on data parallelism, we are developing MASS: a library for multi-agent spatial simulation. It facilitates entity-based programming by allowing an application to be viewed as repetitive interaction among computation entities, each describing independent behavior in its own code, eliminating the need of for-loops to scan all entities. *Places* and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element is called a *place*, and is pointed to by a set of network-independent array indices. Each *place* is also capable of exchanging information with any other *place*s. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place*s with array indices (thus as duplicating themselves), and interact with other *agent*s as well as multiple *place*s.

Temperature prediction can be then parallelized as a MASS application that instantiates a Place object to model an orchard and to maintain temperature for each cell. We can also introduce to this MASS application the effect of air flow or wind that behaves as a collection of agents. In the following two sections, we give further details of this parallelization work.

3 System Overview

Figure 1 illustrates an overview of the orchard temperature-prediction system that we are currently developing in collaboration with AgComm [1] and Valhalla Wireless [11].

Temperature sensor networks are formed with 900MHz long-distance radio devices that work as masters, routers, and slaves [11], all capable of monitoring orchard air temperature. To extend the coverage of monitored areas and stay within budget, we are also planning to use 4.2GHz ZigBee radio devices [1] that are cheaper but limited to short-distance communication, thus functioning only as slaves in need of packet relay via 900MHz router devices to the master node.

We assume that the master node is connected to the Internet for delivering sensor data to temperature-analyzing applications running remotely in clouds. In general, a sensor device transmits its data in UDP packets. However, it is a big burden for application developers to directly manipulate raw UDP packets. To mitigate this burden, we have developed the *Connector* tool kit that facilitates uniform and elastic data channels from sensors to cloud applications, from the applications to mobile users, and between the applications and remote data storages [5]. With *Connector* the master node of each sensor network can behave as a TCP server (termed a *sensor server*), so that temperature-analyzing jobs are blocked until new data is made available.

Our temperature analyzing program consists of the following three components:

1. **Artificial Neural Network**: trains itself with past temperature data downloaded from remote data bases, and thereafter keeps reading real-time sensor data to predict the transition of orchard air temperature for the next three or four hours.
2. **Prediction Polynomials**: reads temperature data sampled everyday at two hours after the sunset, and calculates the overnight temperature transition till the time when the predicted temperature goes down to the lowest degree.
3. **Inverse Distance Weighting**: receives results of per-minute temperature prediction from both the *Artificial Neural Network* and *Prediction Polynomials* components. It then interpolates the results to create temperature maps in a per-minute chronological order.

These components use the *Connector.Graphics* class that provides them with an interface identical to the Java standard but that internally generates JPEG files instead of bitmap images, which will be then sent to a mobile user or a given web server.

Mobile users can use the following two options to check graphical results:

1. **Connector GUI**: launches a graphical menu on remote desktop/laptop computers and allows users to receive JPEG files directly from temperature analyzing programs as well as to control these programs and remote sensor servers.

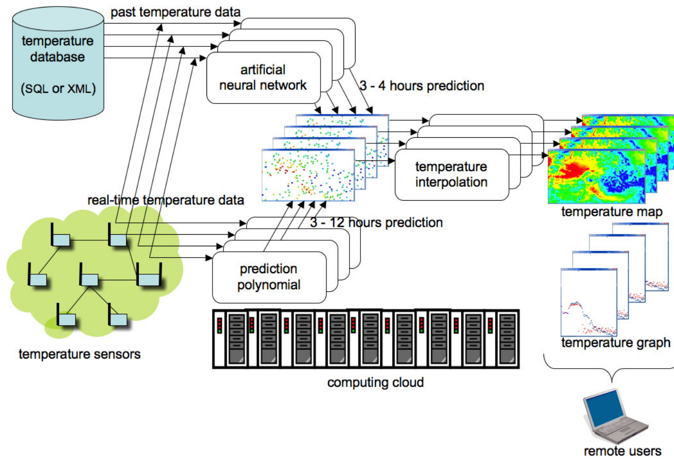

Figure 1. A system structure for orchard temperature prediction

2. **Web Server**: facilitates Connector GUI in the web server, and thus allows users to get access to the remote programs and sensor servers through their mobile devices.

In the next section, we will explain our parallelization strategies for ANN, prediction polynomials, and temperature interpolation.

4 Parallelization

4.1 MASS Library

We define the behavior of an orchard cell and wind flow by extending the *Place* and *Agent* base classes respectively, and then populate them through the *Places* and *Agents* classes. Actual temperature analysis is performed by the following methods.

Places Class

- ***public Places(int handle, String Orchard, Object files, int nFiles)*** instantiates a shared array of *nFiles* elements from the *Orchard* class as passing *files* to the *Orchard* constructor, where each file corresponds to a different sensor.
- ***public Object[] callAll(String predict)*** calls the *predict( )* method of all *Orchard* elements, each corresponding to a different sensor and returning a set of predicted temperature. Calls are performed in parallel among multi-processes/threads.
- ***public void exchangeAll(int handle, String exchangeTemp, Vector<int[]> neighbors)*** calls from each of all *Orchard* elements to the *exchangeTemp( )* of all its neighbors, each indexed with a different *Vector* element. Each vector element, say *neighbor[]* is an array of integers where *neighbor[i]* includes a relative index (or a distance) on the coordinate i from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.

Agents Class

- ***public Agents(int handle, String Windflow, Object files, Places Orchard)*** instantiates a set of *Windflow* agents, passes *files* to their constructor, associates them with the *Orchard* matrix, and distributes them over these places, based on the *map( )* method that is defined within the *Windflow* class.
- ***public void manageAll( )*** updates each *Windflow* agent's status, based on its latest calls of *migrate( )*, *spawn( )*, *kill( )*, *sleep( )*, *wakeup( )*, and *wakeupAll( )*. These methods are defined in the *Agent* base class and will be invoked from other functions through *callAll( )* and *exchangeAll( )*.

At present, we only focus on temperature (but not on wind flow) and therefore use the *MASS.Places* only in our temperature-analyzing programs.

4.2 Artificial Neural Network

The Artificial Neural Network algorithm as designed for temperature prediction uses historical temperature data to train itself and set the appropriate calculation weights to be used for future predictions. This learning period requires that the application process temperature data for the last three months or the same month of the last three years. The weights must be calibrated in order to make accurate predictions in real-time.

The historical data used for our purposes was produced by different sensors and as such exists as separate files for each sensor. Sequential processing of the data within the ANN application is very time consuming and inefficient. To minimize execution time and optimize performance, the processing of each file is designed to occur in parallel using MASS. The MASS process is instantiated to form a grid of ANN places, supporting a one-to-one ratio of a data file to a place. Each sensor's data file is given a unique ID and corresponds to the index of the MASS place that is responsible for using it. When the MASS threads are launched via MASS.callAll(), each place then proceeds to set its prediction engine weights in accordance with

Formulae 4 - 8, and the collective results are integrated upon exit.

After the digestion process the ANN application is ready to make predictions for sensor data obtained in real-time.

4.3 Prediction Polynomials

To obtain more accurate prediction results, a polynomial air temperature prediction algorithm is used to sample temperature two hours after each sunset. The program uses output files provided from temperature sensors through *Connector* data channels. This information is used to predict the temperature of every point in the area from two hours after sunset until sunrise in ten minute increments.

Similarly to the ANN parallelization technique, MASS is utilized to parallelize the execution of the program by creating a grid of polynomial prediction places, one for each sensor, allowing for prediction calculations for all sensor data to be made concurrently. Each input file is also given a unique ID to correspond to the index of the MASS place responsible for its processing. When the MASS service is launched via MASS.callAll() each place uses the polynomial prediction algorithm to calculate its given area temperature in ten minute intervals. Once integrated, the results are depicted in the form of a temperature graph that depicts the change in temperature as a function of time from sunset until sunrise. For real time processing, the results obtained are used to cross check the ANN results for any outliers, as well as make predictions for temperature changes three hours past sunset to sunrise.

4.4 Inverse Distance Weighting

Inverse Distance Weighting (IDW) is used to interpolate current and predicted temperature data and to create a temperature map every time interval, (e.g. 10 minutes in our implementation). We first mesh a given orchard in latitude and longitude precisely enough to have each cell cover all orchard sub-spaces that the crop grower wants to observe. For each time interval, We apply Formula 9 to each cell and thereafter adjust its temperature using its altitude information.

To code IDW with MASS, we instantiate a Places object to represent a meshed orchard where each array element corresponds to a meshed orchard cell. Formula 9 indicates that each cell needs all sensor data for estimating its temperature for an every time interval. MASS can achieve this data distribution by simply invoking *callAll(interpolate, sensor_data)* that passes *sensor_data* to the *interpolate* function of all cells. This function call then starts multiple processes and threads to interpolate each cell's temperature in parallel.

4.5 Output Modeling

The ANN prediction algorithm was shown to have a low margin of error for predictions made only up to 3 hours from sundown. Prediction outliers to and after that point are offset by values calculated by the polynomial prediction algorithm, as they proved to be more in sync with actual results gathered three hours past sunset to sunrise.

Once integrated, the results are depicted in the form of a graph showing temperature changes as a function of 10 minute time increments from sunset to sunrise.

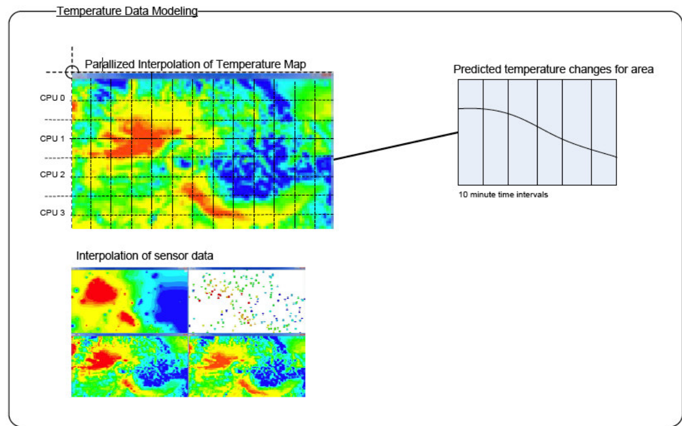

Figure 2. Output modeling

5 Performance Evaluation

5.1 Artificial Neural Network

Application performance was measured on a Dell Precision T5500, running 4 CPUs, each with two cores and two hardware threads, thus allowing for up to 16 threads to execute concurrently. Tests were ran to measure scalability improvements of up to 8 concurrent MASS executions. Measurements are in Milliseconds and show total processing time for each number of data files along with number of CPU cores utilized. Parallelism was observed to improve performance by 63% when processing 8 files, utilizing 4 MASS threads. Due to the hardware configuration of the test computer, MASS thread counts over 4 induced performance degradation, as they perpetuated increased L1/L2 cache misses.

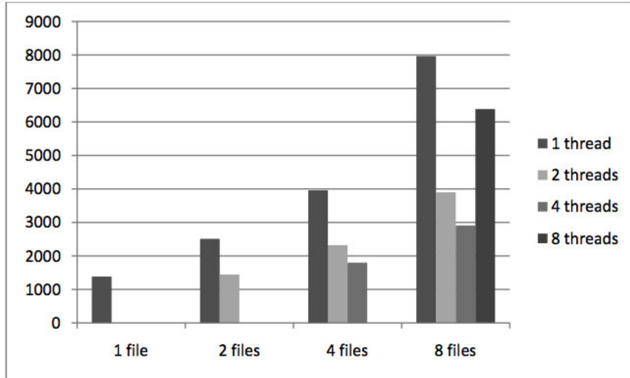

Figure 3. Parallel execution of artificial neural network

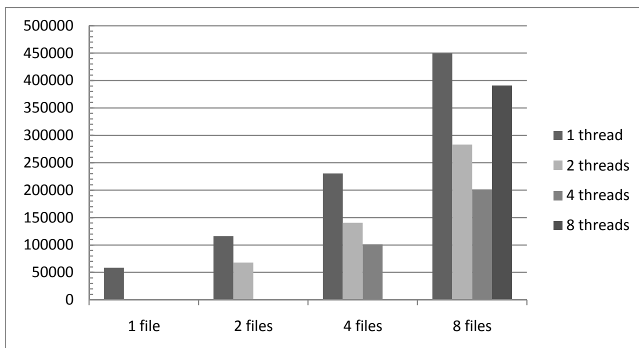

Figure 4. Parallel execution of prediction polynomials

5.2 Prediction Polynomials

Performance testing for the Prediction Polynomial algorithm was done under the same environment conditions as previously mentioned for the Artificial Neural Network. Measurements are in Milliseconds and show total processing time for each number of temperature data files along with number of CPU cores utilized. For performance evaluation, we prepared and processed files that contained interpolated temperature data instead of receiving the data directly from the sensors. Parallelism was observed to improve performance by 55% when processing 8 files, utilizing 4 MASS threads. As mentioned in the previous section, performance degradation was also experienced when utilizing more than 4 MASS threads.

6 Conclusions

This paper described how sensor networks and cloud-computing services can be integrated to predict orchard air temperature for frost-protection purposes. To make available on-the-fly temperature prediction, we are developing the MASS library for multi-agent spatial simulation, parallelized with MASS two temperature-predicting programs: artificial neural network and prediction polynomials. MASS demonstrated competitive parallelization with multiple cores in the same computing node, (up to four in our current experiment). Our next plan is to scale up the computation of these temperature predicting programs over a cluster of multi-core nodes, to parallelize temperature interpolation based on inverse distance weighting, and to develop all the GUI components.

References

[1] AgComm. http://agcomm.net/.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation - OSDI'4*, pages 137–150, San Francisco, CA, December 2004. Publisher.

[3] Fred C. Collins Jr. A Comparison of Spatial Interpolation Techniques in Temperature Estimation. In *Third International Conference/Workshop on Integrating GIS and Environmental Modeling*, pages in CD–ROM, Santa Fe, NM, January 1996. NCGIA.

[4] V. Grimm and S. F. Railsback. *Individual-based Modeling and Ecology*. Princeton Univesity Press, Princeton, NJ, 2005.

[5] J. Melchor and M. Fukuda. A design of flexible data channels for sensor-cloud integration. In *Proc. 21st International Conference on System Engineering - ICSEng 2011*, page to appeaer, Las Vegas, NV, August 2011.

[6] Message Passing Interface (MPI) Standard. http://www.mcs.anl.gov/research/projects/mpi/.

[7] OpenMP Specifications. http://www.openmp.org/mp-documents/spec30.pdf.

[8] A. P. C. Services. Market applications frost & freeze protection. Web page, http://www.agroshield.com/market-applications.html, 2007.

[9] B. A. Smith, G. Hoogenboom, and R. W. McClendon. Artificial neural networks for automated year-round temperature prediction. *Computers and Electronics in Agriculture*, Vol.68(Issue.1):52–61, August 2009.

[10] UC Davis Biometeorology Program - Frost Protection. http://biomet.ucdavis.edu/frost-protection.html.

[11] Valhalla Wireless. http://valhalla-wireless.com/vwr/.