Experience with Lamport Clock Ordered Events with Intel Threading Building Blocks in a Glucose-Level Prediction Software

Tomas Koutny

Department of Computer Science and Engineering
University of West Bohemia
Univerzitni 8, Plzen 306 14
Czech Republic

txkoutny@kiv.zcu.cz

Abstract. Software tool was needed to verify a model predicting interstitial fluid glucose level, while conducting an experiment. With the tool, several tasks execute concurrently to effectively utilize available processors. Implementing the tool implied addressing such aspects of parallel computing which possibly have a broader impact. In this paper, I present an experience with implementing Lamport-clock ordered event scheme to control a parallel program employing a task-stealing scheduler, while eliminating the possibility of accidentally masking a synchronization error. For a program based on Intel Threading Building Blocks library, I devised a scheme to control task execution with events. These events are ordered using the concept of Lamport Clock. As the causal ordering of events is complete, program's behavior can be reconstructed for additional debugging. In the implementation devised, recording the events induces no additional synchronization operations that could accidentally mask a synchronization error. The work is presented in a context of glucose-level prediction that originates from a glucose-transporter research.

Keywords: programming paradigms, parallel systems, debugging aids, glucose level prediction

1 Introduction

Glucose is distributed throughout the body primary through the blood vessels. The maintenance of a normal blood glucose level is accomplished by a network of hormones, neural signals, and substrate effects that regulate the endogenous glucose production and the glucose utilization by tissues other than the brain [1]. From the blood, glucose is transported through the blood capillary membrane to the interstitial fluid. The interstitial fluid, which is found in the intercellular spaces between tissue cells, supplies the cells with nutrients, including glucose. In the interstitial fluid, the glucose is either utilized or leaves the interstitial fluid to eventually return to the blood. The lymphatic system represents an accessory route through which the fluid can flow from the interstitial spaces into the blood [2].

As the body regulatory mechanisms try to maintain blood glucose level within a particular range, the glucose homeostasis can be modeled. A particular model was proposed to predict glucose level in various compartments such as subcutaneous tissue, skeletal muscle tissue and visceral fat [3, 4, 5]. A software tool was developed to verify the model, while conducting an experiment. The tool executes several tasks concurrently to effectively utilize available processors. Implementing the tool implied addressing such aspects of parallel computing which possibly have a broader impact. In this paper, I present an experience with implementing Lamport-clock ordered event scheme to control a parallel program with a task-stealing scheduler, while eliminating the possibility of accidentally masking a synchronization error.

1.1 Prediction Model

Equation (1) gives the particular prediction model proposed. It relates present blood and interstitial fluid glucose levels to future interstitial fluid glucose level [3]. The *b(t)* and *i(t)* symbols denote the respective blood and interstitial fluid glucose levels at the time *t*.

$$p \times b(t) + cg \times i(t) \times (b(t) - i(t)) + c = i(t + \Delta t) \tag{1}$$

The Δt, *p*, *cg* and *c* symbols are supposed to have the following meaning [3, 6]. Δt denotes the prediction interval. When compared to Fick's Law of Diffusion [7], *b(t)-i(t)* is the concentration difference across a membrane, and *cg* is the surface area, multiplied by membrane permeability, and divided by the thickness of the membrane. Accounting the diffusion back across the membrane, *p* expresses a glucose gain from the blood, thanks to intercellular clefts between the endothelial cells. *c* is an arbitrary glucose level that covers the difference between the flux from the blood into the interstitium and the flux from the interstitium into the cells.

As the work on the model progresses, several modifications of the model execute concurrently. It is desirable to see the difference between the very Equation (1) and its modifications. For example, boundary conditions were proposed for Equation (1) to study glucose predictability, blood capillary permeability and glucose utilization rate [6]. Another experimental feature is to expand the *c*-parameter into a model describing the function of GLUT/SLC2A family of proteins. These proteins are glucose transporters, which mediate a facilitated diffusion of glucose into muscle and adipose cells [8]. Another aim is to reverse the model (1) into a blood glucose level reconstruction model and to compare it with the present plasma-interstitium kinetic model [9]. When new glucose levels are measured during the experiment, they have to be either approximated [5] or interpolated [10] prior doing these calculations. Together, all these calculations are tasks, which should execute concurrently as much as possible.

1.2 Experiment

Based on the similarity in sugar and insulin physiologies between humans and rats, experimenters are able to conduct the required experiments on rats. The work presented was tested on hereditary hypertriglyceridemic rats. The rats were provided by the Diabetology Center, University Hospital in Pilsen, Charles University in Prague, and the experiments were conducted by researchers from this institution.

First, the experimenter administered a combination of xylazine and ketamine as an anesthetic. The specific chemicals used were xylazine (active ingredient, xylazine hydrochloride) and Narkamon (active ingredient, ketamine hydrochloride), which are drugs that are manufactured by Bioveta a.s.

The experimenter then catheterized the internal jugular vein and the carotid artery of the anesthetized rats. The blood glucose level was measured in the arterial blood. Sensors associated with CGMS were placed in the subcutaneous tissue, skeletal muscle tissue, and abdominal subcutaneous tissue, i.e., the visceral fat. After the sensors were calibrated, insulin infusion was started. The insulin infusion rate was constant at 50 mUI/kg/min. A variable 20% glucose infusion was also started using a manual correction to maintain the desired blood glucose level of 6 mmol/l. The rats were administered the insulin and glucose infusions through their jugular vein.

After 15 min at this steady state, the experimenter administered a bolus of 0.5 g/kg glucose. The experimenter then attempted to reach a new steady state with a blood glucose level of 12 mmol/l. After 60 min, the experimenter administered a bolus of 0.5 UI/kg short-action insulin and stopped both infusions. The experimenter continued to monitor the glucose levels for an additional 80 min. At the end of the experiment, the animal was sacrificed.

The glucose levels of all of the compartments were measured simultaneously every 5 min. The measurement tolerance for the blood glucose level was ±0.2 mmol/l. The CGMS measurement tolerance was 15%. Medtronic Guardian® REAL-Time was the CGMS used.

For such an experiment setup, let us illustrate performance of the model with Table 1 [6]. In Table 1, the p and cg parameters are dimensionless. The unit of the c parameter is [mmol/l]. The study calculates the prediction error as the difference between the calculated interstitial fluid glucose level and the following:

1. measured interstitial fluid glucose levels
2. and approximated measured interstitial fluid glucose levels [5].

The prediction error, which was in units of [mmol/l], was calculated as the average absolute difference and as the maximum absolute difference. The first number shown in Table 1 is the median value. The numbers within the brackets are the first and third quartiles, respectively.

Table 1. Calculated Quantities, i.e., the Equation (1) Parameters and the Reaction Delay

Compartment / Quantity	Subcutaneous Tissue	Skeletal Muscle Tissue	Visceral Fat
Glucose Gained from the Blood (p Parameter)	0.971 (0.874; 0.992)	1.000 (0.773; 1.000)	1.000 (0.979; 1.000)
Effect of the Membrane Surface Area and Permeability (cg Parameter)	-6.217 (-6.722; -5.374)	-5.298 (-6.351; -4.424)	-5.921 (-6.102; -5.028)
Average Residual Mass of Glucose (c Parameter) [mmol/l]	1.127 (0.432; 1.812)	0.420 (0.245; 1.883)	0.810 (0.487; 0.978)
Prediction Interval [min:sec]	20:00 (15:30; 21:15)	8:30 (5:00; 20:00)	5:58 (5:00; 7:01)
Approximation Average Error [mmol/l]	0.336 (0.279; 0.476)	0.422 (0.311; 0.636)	0.373 (0.334; 0.476)
Approximation Maximum Error [mmol/l]	1.423 (0.759; 1.610)	1.363 (1.250; 1.618)	1.382 (1.158; 1.697)
Measured Average Error [mmol/l]	0.363 (0.287; 0.498)	0.513 (0.350; 0.634)	0.460 (0.377; 0.565)
Measured Maximum Error [mmol/l]	1.011 (0.712; 1.647)	1.966 (1.555; 2.173)	1.818 (1.526. 2.772)

1.3 Intel Threading Building Blocks

The prediction program solves a computationally intensive task, when it calculates parameters of the prediction model (1) and other models implemented. The prediction of the interstitial-fluid glucose level has to be completed in less than two minutes, if it should affect the glucose infusion rate. For a parallel algorithm used in applied bioinformatics, synchronization operations may become a bottleneck when processing a large data set, or when reducing the total execution time. Therefore, this work is possibly interesting to biomedical researches as it presents a useful synchronization scheme – an addition to a library designed for developing calculation-intensive applications.

The glucose-level predicting program is decomposed into the user-interface frontend and the prediction-calculating backend. The backend abstracts the processor time with tasks, not threads. It uses the Intel Threading Building Blocks library [11]. This library presents a high-level task-based parallelism abstraction. The entire calculation is partitioned to smaller tasks, which are scheduled efficiently to available cores using the task-stealing algorithm [12] while reducing the adverse effects of cache-cooling, context switching between logical threads and lock preemption [13]. The cache-cooling refers to a scenario, when a processor had to evict items from a cache to pull them back later (e.g. due to a context switch), but at the cost of hundreds of cycles per each cache miss [14].

The library presents the parallel-programming paradigm that defines tasks which run in shared memory. The library separates logical tasks from physical threads, thus it scales well on multi-core chips. A programmer decomposes the problem into a set of tasks, while expressing their mutual dependencies. Then, the library executes the tasks using such a number of logical threads, which corresponds with the number of physical threads.

2 Event Scheme

The Intel Threading Building Blocks library offers a cancellation mechanism to control execution times of the tasks. For the prediction program, I devised event-based scheme that allows a finer-grained control over the tasks than the present cancellation mechanism does.

Glucose level prediction is calculated using a set of parameters. In the practice, these parameters can be updated after some glucose levels are already predicted. However, it is not always necessary to discard the already predicted levels. Instead of this, it is possible to continue with the prediction once new parameters are obtained. The present cancellation mechanism of the Intel Threading Building Blocks library does not allow this. Therefore, a different mechanism to control the calculation was needed.

The inspiration came from MS Windows messages, IDataAdviseHolder and IAdviseSink interfaces of the Component Object Model (COM) [15]. The reason was to follow a well-known programming practice. Therefore, data structures and interfaces devised may look familiar to COM programmers, but they are different. There are three different objects with the following chain of event flow:

```
Sender → Holder → Receiver 1 … Receiver n
```

To fire an event, the sender calls the holder object using holder's SendOnEvent method. To deliver the event, the holder calls OnEvent method of each receiver that is connected to the holder.

To receive events, receiver-object's class implements IEventAdviseSink interface with a single method – void IEventAdviseSink::OnEvent(EVENTMEDIUM *event). On event, this callback is invoked and a valid pointer is passed. The structure is declared as follows.

```
typedef struct _EVENTMEDIUM {
  void *Sender;
  int Clock;
  int Code;
  size_t wParam;
  size_t lParam;
} EVENTMEDIUM, *PEVENTMEDIUM;
```

The Sender member identifies the sender. This-pointer can be passed easily as the sender is usually an object. The clock is the Lamport Clock of the program. Code identifies the event; wParam and lParam pass additional information about the event. The receiver is supposed to merely notice the asynchronous event and to process it later – synchronously to its own calculation.

The Lamport Clock [16] was designed to determine order of events in a distributed system. As clocks of different components of the distributed system might not be synchronized, an incrementing integer counter is used per each component – the Lamport Clock. The mechanism allows capturing the happened-before ordering numerically. For the following two reasons, I consider use of the Lamport Clock as better than relating the event time to a coordinated time:

- Incrementing an integer counter atomically presents no overhead when compared to getting system time from the operating system. Incrementing the integer counter avoids a possible synchronization operation with a different thread as such a thread may compete for the same resource. For an open-source example, the current_kernel_time function must be called to get the current time on Linux. This function synchronizes with sequential lock [17]. If no write occurs to the system time variable, two concurrently reading threads will not be affected. They will return from the function in the same order, in which they called the function if none of them was preempted. Serving an interrupt, e.g. the clock-generated one, increases the chance that one of these threads will be preempted. Thus, there is a chance of altering the order in which the threads will return from the function. As subtle as such a chance is, using a single atomic instruction avoids it.

- Reading a processor’s time-stamp counter can set the EVENTMEDIUM.Clock member, using the rdtsc instruction on x86. However, the CR4 register controls whether this instruction can only be executed at privilege level 0 [18]. On contrary, it is possible to increment an integer counter atomically at any privilege level. Therefore, integer counter is a better choice as the x86 is the present target processor.

To start receiving events, the object with the IEventAdviseSink interface implemented has to register with an object with the IEventAdviseHolder interface implemented.

```
class IEventAdviseHolder {
public:
  HRESULT Advise(IEventAdviseSink *Sink,
                 int *Connection) = 0;
  HRESULT Unadvise(int Connection) = 0;
  HRESULT SendOnEvent(int Code,
                      size_t wParam, size_t lParam) = 0;
  HRESULT NameTheHolder(wchar_t *name) = 0;
};
```

The Advise method sets up a notification connection to the receiver. The Unadvise method destroys the connection. The SendOnEvent method fires the event to all connected receivers. The NameTheHolder method associates a meaningful name with the holder to ease the debugging when printing the events recorded. The HRESULT data type can be defined easily with typedef on non-Windows platforms.

3 Events’ Implementation

The number of instantiated objects, whose classes implement the IEventAdviseHolder interface, is not limited. They only share a single integer counter that represents the Lamport clock. As the program uses the Threading Building Blocks library, the clock is declared as tbb::atomic<int>. The clock is initialized to zero. The fetch_and_increment() function increments the clock and returns the old value.

In each class implementing the IEventAdviseHolder interface, there is a class member declared as std::vector<EVENTMEDIUM>. As the holder object processes its events, the events are pushed to this vector. Thus, events are stored in a decentralized manner in several objects. Thus, recording the events causes no additional synchronization. In the implementation, all these objects are long-life ones. When the holder object is deleted, it prints the events recorded to the debug output.

Along calculation-specific event codes, there are five debugging event codes: ecAdvised, ecUnadvised, ecSendOnEventEnter, ecSendOnEventExit and ecSendOnEventFailed.

When a receiver connects to an event holder, the holder increments the clock and records this as an event (ecAdvised). The event stores connection identification (in wParam) and receiver-object's identification (in lParam). As the receiver is required to implement the IEventAdviseSink interface, the receiver-object's identification is address of this interface implementation. Similarly, when disconnecting, the holder increments the clock and records this as an event (ecUnadvised). The event stores identification of the connection dropped (in wParam).

When a sender object calls a holder to deliver a particular event, the holder generates and records two additional events. First, the holder increments the clock and records this as an event (ecSendOnEventEnter). The event records that a particular sender requested a particular holder to deliver an event. Then, the holder increments the clock and delivers the event to each connected receiver. Finally, the holder increments the clock and records this as an event (ecSendOnEventExit). It records that the event was fired and delivered to all connected receivers successfully. In a case of failure, an event indicating the failure is recorded instead (ecSendOnEventFailed). In all these three cases, the number of connected receivers is recorded (in wParam).

As a result of these rules described, the recorded events provide the following information:

1. The number and identifications of receivers connected to each particular holder is known.
2. It is known which sender fired which event.
3. Based on the items 1 and 2, it can be determined which event was delivered from which sender to which receivers.
4. Nested events are detected due to the ecSendOnEventEnter and ecSendOnEventExit/Fail events.
5. It is detected when event delivery failed and if the number of receivers changed while the event was being fired.
6. The causal ordering of events is complete. For any two events, it is always known which event happened before.
7. Due to the ecSendOnEvent enter and exit events, events being fired concurrently are detected.
8. In addition, the event scheme and its implementation are compatible with the vector clock of distributed systems.

Generating the debug events is a subject to a conditional compilation of the event library. A release version can omit the debug events, unlike a debug version. When compiled as a dynamic library, the debug or release version can be loaded by the program as needed. To the debug output, the senders and receivers can print meaningful names next to their identifications, which are recorded by the holder objects.

4 Controlling tbb::task with Events

When implementing, the Threading Building Blocks' task inherits from the tbb::task. The inherited task must override the tbb::task* tbb::task::execute() method. This method implements the task's activity. During the activity, the task may create and wait for other tasks as needed. Once finished, the task returns either NULL or a pointer to a task that executes after this task has finished. Entire calculation finishes when all tasks have finished. A task cannot be terminated. It can be merely asked to cancel its activity gracefully. A cancelled task skips the execute method, if it has not started yet. Otherwise, the cancellation has no effect. Just the task can poll tbb::task::is_cancelled bool [6].

In the implementation of the event scheme devised, there is a thread that begins the calculation. It spawns tbb::tasks, which begin the calculation. The thread is wrapped with an object. Internally, this object is called MasterCalculation. It receives events from the program's frontend to control the program's backend. The thread-wrapping, MasterCalculation, object signalizes the events received to the tasks.

The event scheme devised applies to a calculation with a fixed number of tasks. All tbb::task objects are allocated before the actual calculation begins. Each tbb::task object has an unsigned integer property signalizing pending events. Each event has a corresponding bit. On event, the MasterCalculation object sets the respective bit for each tbb::task object with an atomic operation – OR particularly. After doing a certain amount of work, the tbb::task object checks the bits set to react on pending events. The event-parameters are not propagated to the tbb::task objects. With multiple events of the same type, the tbb::task object reacts to a particular type of event just once. This eliminates the overhead of reacting to an outdated event. Furthermore, it enforces a synchronous event-processing from the tbb::task object perspective. Such a processing eliminates synchronization-related overhead and possible race conditions that would be induced otherwise by an asynchronous event processing. The following code fragment illustrates this.

```
#define pefCancelCalculation  1
   //The tasks should terminate gracefully.
#define pefParametersChange   2
   //The parameters have changed and should be updated.

class CTask : public tbb:task {
protected:
  tbb::atomic<unsigned int> mPendingEvents;
public:
  void SignalEvents(unsigned int NewEvents) {
    AtomicOr(&mPendingEvents, NewEvents);
  }
```

```
 void SetupParameters {
   //As the event parameters are not propagated to tbb::
   //task objects, current parameters must be obtained.
 }

 tbb::task* Execute() {
   //Run until a cancellation is requested.
   while (!(mPendingEvents & pefCancelCalculation)) {
     //Serve all signalized events,
     //e.g. parameters change.
     if (mPendingEvents & pefParametersChange) {
       AtomicAnd(&mPendingEvents, ~pefParametersChange);
       SetupParameters();
     }

     //If there is no event signalized, then calculate.
     while (!mPendingEvents) {
       DoAPieceOfWork();
     }
   } //while (!(mPendingEvents & pefCancel...
  } //tbb::task* Execute

}; //CTask

class CMasterCalculation : public IEventAdviseSink {
protected:
  tbb::task* mTasks[TaskCount];
  tbb::atomic<unsigned int> mPendingEvents;
public:
  void OnEvent(EVENTMEDIUM *event) {
    AtomicOr(&mPendingEvents, event->Code);
    for (size_t i=0; i<TaskCount; i++)
      mTasks[i]->SignalEvents(event->Code);
    }

  void Execute() {
    tbb::task_list list;
    do {
      WaitForASignalToStartTheCalculation();
        //Waits until the program's frontend signalizes
        //with a conditional variable.
      list.clear();
      list.push_back(...);
      spawn_root_and_wait(list);
        //En-queues first tasks, which can run
        //and waits until all tasks have finished.

      SendOnEvent(this, ecDataCalculated, 0, 0, 0);
        //The program's fronted is connected to this
        //object. It receives this event once
        //the calculation finishes.

    } while (!(mPendingEvents & pefTerminate));
  }
}; //CMasterCalculation
```

5 Conclusion

The presented event-based scheme was verified with a particular implementation. The implementation calculates glucose quantities for the blood, subcutaneous tissue, the skeletal muscle tissue and the visceral fat [3, 4, 6, 10]. In addition to the debugging events, the following events are used to control the calculation.

- pefTerminate – This event signalizes that the program terminates so that the MasterCalculation object should finish its execution. On this event, the MasterCalculation object signalizes the pefCancelCalculation to each tbb::task object and waits until they all finish.
- pefCancelCalculation – A tbb::task object detecting this event should cancel its calculation gracefully, yet immediately.
- pefDataAvailable – When new data are measured, the frontend generates this event. To process all measured data, the MasterCalculation object runs the calculation repeatedly as long as this event flag is set.
- pefRecalculateAll – All running tbb::tasks cancel their activity, discard the already calculated results and the entire calculation is restarted.
- pefParamatersChange – This event assumes a situation, when the user changes some parameters, but it is not necessary to restart the entire calculation. All running tbb::tasks just fetch the recent calculation parameters and continue the calculation.

Evaluating the scheme from a practical experience [3, 4, 6, 10] in the horizon of two past years, it provides all the functionality needed. The practice gave no need to modify the scheme devised.

6 References

1. Longo, D., Fauci, A., Kasper. A., Hauser, S., Jameson, J., Loscalzo, J.: Harrison's principles of internal medicine. Mc Graw-Hill, New York (2011)
2. Guyton, A.C., Hall, J.E.: Medical textbook of physiology. Elsevier Inc., Philadelphia (2006)
3. Koutny, T.: Prediction of interstitial glucose level. IEEE Trans. Inf. Technol. Biomed. 16, 136-142 (2012)
4. Koutny, T.: Estimating reaction delay for glucose level prediction. Med. Hypotheses 77, 1034 – 1037 (2011)
5. Koutny, T.: Modeling of compartment reaction delay and glucose travel time through interstitial fluid in reaction to a change of glucose concentration. In 10th IEEE International Conference on Information Technology and Applications in Biomedicine, Corfu (2010)
6. Koutny, T.: Glucose Predictability, Blood Capillary Permeability, and Glucose Utilization Rate in Subcutaneous, Skeletal Muscle, and Visceral Fat Tissues. Comput. Biol. Med. 43, 1680-1686 (2013)
7. Bronzino, J.D.: The biomedical engineering handbook. CRC Press, Connecticut (2006)
8. LeRoith, D., Olefsky, J.M., Taylor, S.I.: Diabetes Mellitus: A fundamental and clinical text. Lippincott Williams & Wilkins, Philadephia (2003)
9. Facchinetti, A., Sparacino, G., Cobelli, C.: Sensors & algorithms for continuous glucose monitoring reconstruction of glucose in plasma from interstitial fluid continuous glucose monitoring data role of sensor calibration. J. Diabetes Sci. Technol. 1, 617-623 (2007)
10. Koutny, T.: Gluocose-level interpolation for determining glucose distribution delay. In XIII Mediterranean Conference on Medical and Biological Engineering and Computing, Sevilla (2013)
11. Intel® Threading Building Blocks Reference Guide. Intel Corporation Document Number: 315415-015US
12. Lu, S.: Improving the task stealing in Intel threading building blocks. In International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Beijing (2011)
13. Intel® Threading Building Blocks Documentation. Intel Corporation Document Number: 327304-002US
14. Reinders, J.: Intel threading building blocks: Outfitting C++ for multi-core processor parallelism. O'Reilly Media, Sebastopol (2007)
15. Microsoft Developer Network IDataAdviseHolder interface (COM). http://msdn.microsoft.com/en-us/library/windows/desktop/ms686622%28v=vs.85%29.aspx
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. CACM 21, 558-565 (1978)
17. Love, R.: Linux Kernel Development. Addison-Wesley Professional, Crawfordsville (2010)
18. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation Order Number: 325462-045US