Performance analysis of the parallel code execution for an algorithmic trading system, generated from UML models by end users

Gaétan Hains, Chong Li, Nicholas Wilkinson, Jarrod Redly, Youry Khmelevsky

2015 National Conference on Parallel Computing Technologies (PARCOMPTECH). IEEE, 2015. p. 1-10.

Performance Analysis of the Parallel Code Execution for an Algorithmic Trading System, Generated From UML Models by End Users

Gaétan Hains*, Chong Li†
Lab. d'Algorithmique, Complexité et Logique
Université Paris-Est Créteil
Paris, France
Emails: gaetan.hains@gmail.com, research@chong.li
*Also affil. with LIFO, Univ. of Orleans, France.
†Also affil. with National Inst. of Informatics, Tokyo, Japan.

Nicholas Wilkinson, Jarrod Redly, Youry Khmelevsky
Computer Science Department
Okanagan College
Kelowna, BC, Canada
Emails: kharnaxin@gmail.com,
jarrod_redly@hotmail.com, ykhmelevsky@okanagan.bc.ca

***Abstract*—In this paper, we describe practical results of an algorithmic trading prototype and performance optimization related experiments for end-user code generation from customized UML models. Our prototype includes high-performance computing solutions for algorithmic trading systems. The performance prediction feature can help the traders to understand how powerful the machine they need when they have a very diverse portfolio or help hem to define the max size of their portfolio for a given machine. The traders can use our Watch Monitor for supervising the PNL (Profit and Loss) of the portfolio and other information so far. A portfolio management module could be added later for aggregating all strategies information together in order to maintain the risk level of the portfolio automatically. The prototype can be modified by end-users on the UML model level and then used with automatic Java code generation and execution within the Eclipse IDE. An advanced coding environment was developed for providing a visual and declarative approach to trading algorithms development. We learned exact and quantitative conditions under which the system can adapt to varying data and hardware parameters.**

***Keywords*-UML; code generation; high performance computing; BSP; performance prediction; parallel programming; algorithmic trading**

I. Introduction

Algorithmic trading systems that are widely used by many institutional organizations or individual traders play a more and more important role in our global economic system to manage market impact, risk, and to provide liquidity to the market. A professional trader need to diversify his/her portfolio with different trading strategies (i.e. algorithms) in different markets with different stocks in order to limit the risk in a controllable level. This is a computation-intensive and time-critical work for today's markets. The most common way is that the quantitative analysts (Quant) works in pairs with software (SW) engineers for coupling trading algorithms design, development and deployment. Therefore, Quant handles the financial aspect and SW engineer deals with computer performance and code quality.

However, a Quant would not always find his ideal SW engineer for solving a complex algorithm in an high performance way. Facing to this issue, a first step was proposed by Li and Hains [2], [3] using the SGL software-hardware bridging model to simplify the parallel development. Inspired by SGL's performance prediction feature and divide-assign-bridge philosophy, we propose in this paper an adaptive code generation as a second step for raising the simplicity from parallel program development to algorithmic trading design. We describe here an algorithmic trading prototype and propose performance optimization of end-user code generation from customized UML models. Our prototype includes high-performance computing solutions for algorithmic trading systems. The performance prediction feature can help the traders to understand how powerful the machine they need when they have a very diverse portfolio or help hem to define the max size of their portfolio for a given machine.

The prototype can be modified by end-users on the UML model level and then used with automatic Java code generation and execution within the Eclipse IDE. An advanced coding environment was developed for providing a visual and declarative approach to trading algorithms development.

The initial "Algorithmic Trading System" prototype (Prototype) was developed within a Software Engineering Project course at University of British Columbia Okanagan Campus (UBC O) in Canada, in collaboration with the international student research project sponsors, Laboratoire d'Algorithmique, Complexité et Logique (LACL) in Université Paris-Est (UPE) in France. It was developed as a new approach for special-purpose programming language code generation from UML models by designing and programming a new plugin for Eclipse Modeling Tools (Indigo SR2) IDE, and a developed library of predefined functions for the plugin [4]. Not all experiments, related to parallel code execution and performance optimization were finished in the Prototype by UBC O students in 2012, because of lack of time and some other issues. In current paper students of COSC 470 SW Engineering Capstone Project course at Computer Science Department, Okanagan College with support from UPE finished implementation of bulk-synchronous parallel algorithm (BSP) and

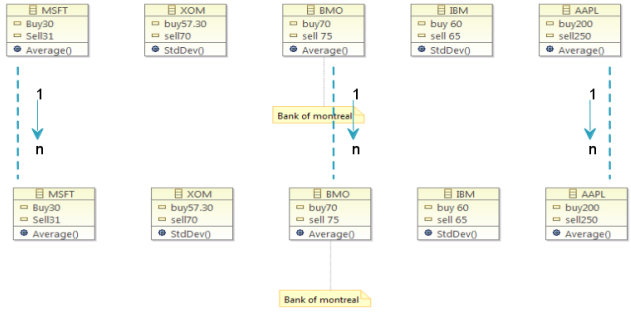

Fig. 1. End-User Customizable UML Diagram of the On-line Stock Information Sources.

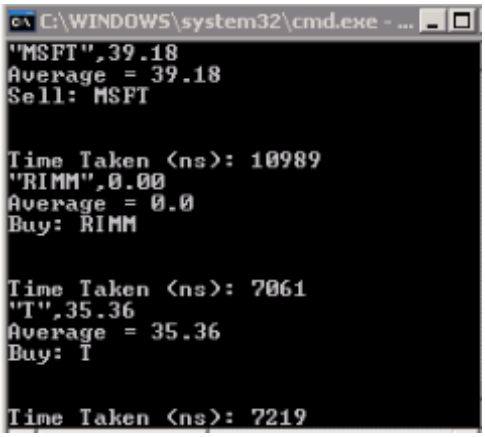

Fig. 2. Console output.

conducted experiments to prove performance optimization for the Prototype.

The goal in developing the Prototype was to enable end-users to modify the software application without programming or support from programmers. Additional goal was to improve performance of the application by implementing adaptive code generation, depends on architecture of the computer (number CPUs and cores). Current code generation tool was built upon with respect to functionality requested by clients for a hedge fund in Paris. The Prototype allows end-users to create simple business related UML diagram (see a UML diagram example in Fig. 1) and to generate a Java code used in calculating moving averages [6] in text mode within operating system (OS) console (see Fig. 2) and in graphical mode (see Fig. 3). The Eclipse Modeling Framework (EMF) [9] and Java programming language were used for the Prototype. The tool can be run in both modes by an end-user in an OS console without EMF, but EMF is required to update UML model (configuration of the tool) and to regenerate java code by the developed plugin from graphical user interface of the EMF (Fig. 4).

Our main contributions include: 1) a visual and declarative approach that allows end-user (Quant) create parallel programs (algorithmic trading strategies) without strong coding skills; 2) a software prototype that proves the feasibility of fast generating error-less code; 3) an auto-adaptive approach that

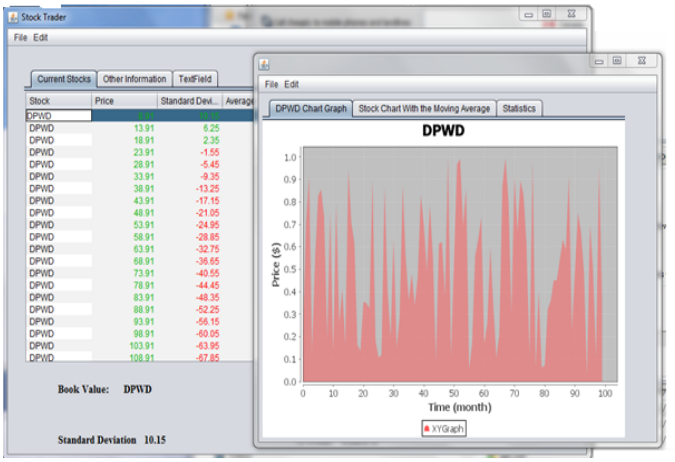

Fig. 3. Real-Time Stock Watch Monitor with the Graphical Stock Information Diagram.

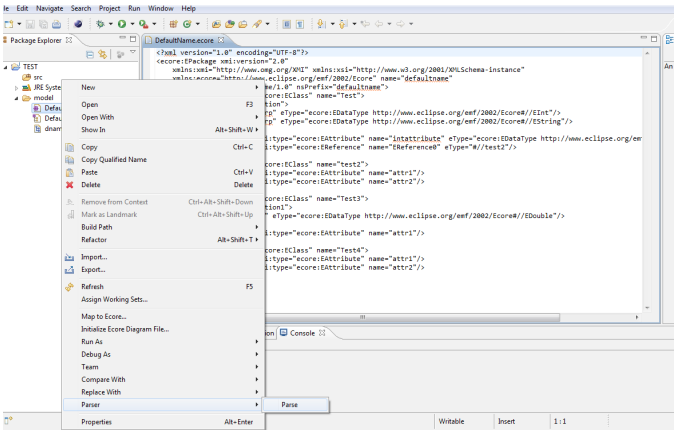

Fig. 4. Code Generation Plugin.

generated parallel system can obtain the best performance by adapting to varying data and hardware parameters.

The rest of this paper is organized as follows: In Section II we discuss the original system design with the information about our research background very briefly, in Section III we discuss performance optimization and related issues, in Section IV we discuss our resent testing results (Fall 2014) and BSP code implementation for the Prototype, in Section V we discuss related research papers and implementations, in Section VI we summarize our current research results, and in Section VII we discuss our future work.

II. Code Generation System Design

UML diagrams were used for the front-end and Java as the back-end to generate executable code directly from high-level specification. Investigation into specifying algorithmic trading strategies using UML business models (see Fig. 1), model-driven development [13], and automatic code generation for general purpose programming languages was carried out [4], [6].

The resulting Prototype is capable of generating code from UML business models and predefined Java libraries; code

Fig. 5. Example of the .ecore XML file.

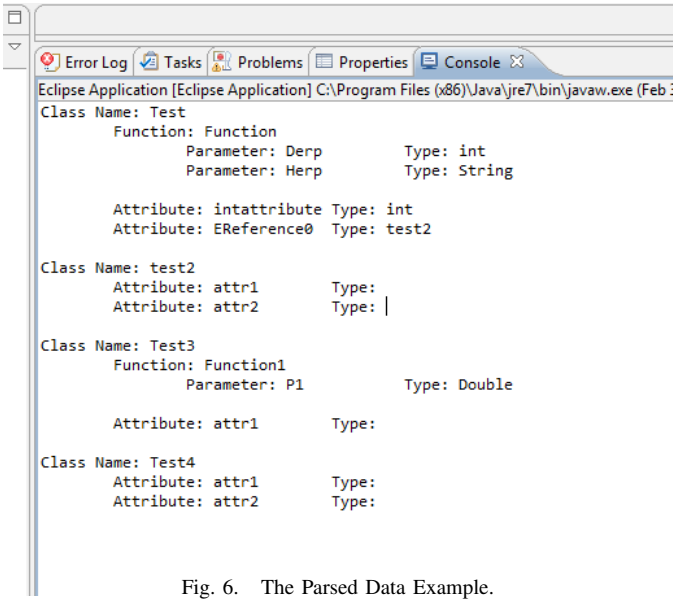

Fig. 6. The Parsed Data Example.

generation is used to produce executable code from models, as discussed in [13].

A Java code-generating Prototype takes information from a UML diagram (see Fig. 1) and creates real-time Java applications, supporting parallel processing [4], [6]. EMF is used to create a specification for the plugin by using UML models. Using the EMF plug-in the Prototype is able to produce a UML diagram of the specifications, which is stored as an XML file known locally as an .ecore file (see an example in Fig. 5). The relevant information is extracted from the .ecore file by using a simple Java parser, which gathers the required methods from the library and adds them to the generated output. The parsed data example is shown in Fig. 6. The code generator then calls the parser which takes UML data and creates the Stock Watch Monitor Java application (Stock Watch). The Stock Watch application collects data from Yahoo in real-time and produces a table with the data results, as shown in Fig. 3. More information about the first Prototype implementation can be obtained from the project documentation [6] and from [4].

III. Optimization by code generation

Having demonstrated a small but realistic system based on our ”no-programming” paradigm, we wish to extend its adaptive features to non-functional properties like performance.

As a first step in this direction, we have conducted experiments in measuring performance to learn exact and quantitative conditions under which the system can adapt to varying data and hardware parameters. In the first experiment, we show how financial-data stream processing can scale to a variable number of incoming streams and/or a variable number of hardware processing elements. Globally the performance model is a linear relationship between processing rate, number of cores and number of incoming streams. But the experiments also reveal exact threshold values related to real-time processing and Java garbage collection. For each parameter we describe a method whereby our automatic code generation can adapt a priori to optimize performance. We also outline the scheme's adaptation to distributed-memory parallel processing in the presence of multiple streams and multiple processing elements.

A. Initial experimental setup

The performance experiments we have conducted consisted in processing streams of real-time financial data from Yahoo finance (dated June 2012) incoming at the rate of 1 record per second. Each input stream is a sequence of quotation values for one stock “symbol” among the following: GE, INTC, YHOO, NVDA, ZNGA, EBAY, MU, ORCL, GOOG, MSFT, RIMM, DELL, RY, BNS, CSCO, SIRI.

An automated trading strategy will generally process incoming data one at a time, accumulate some statistics, and issue orders BUY, SELL, or NIL. We have generated test “strategies” that compute a standard deviation on the stock value and, to balance the very-low frequency of incoming data (HFT or High-frequency trading often runs at maximal rates of hundreds of input-order loops per second), we have artificially repeated the computation a large number of times within each input-output loop. Each loop's computational result is of no importance but it represents a typical computation in HFT since trading automata are often built from decision trees using sliding statistics on the input stream(s).

The generated Java code was run on a Samsung portable computer with a 2-core Intel i5 processor running at 1.8GHz with 4GB of RAM and Windows 7 Pro as 64-bit O.S.

B. Performance data and analysis

Each performance experiment takes as input between 1 and 16 streams, each one corresponding to a particular stock symbol. One processing thread has been generated to process each input stream, as described above and without synchronization between them.

For each thread and at each loop we have measured the accumulated system time T_{acc} in ms since the beginning of execution. Data is buffered locally so there is no internet transport delay to account for. The initial record is already loaded when computation starts and it is assumed that a new one arrives every second. The real-time constraint for each

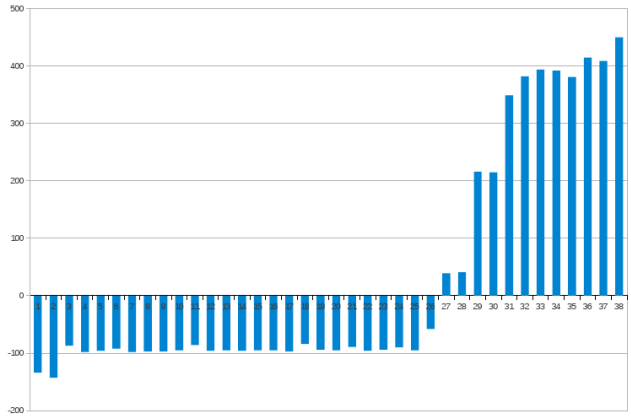

Fig. 7. Overtime vs Ticks.

thread is thus to maintain $T_{\text{acc}} < (1000\text{ms}) \times n$ where n is the number of loop runs since the start. The application-specific performance value we have computed is called *overtime* and is computed as $T_{\text{acc}} - (1000\text{ms}) \times n$. In other words the experiments will measure the conditions for keeping overtime negative i.e. staying within real-time operations where the process issues market orders no slower than market data is received.

For example the 1-symbol (one financial instrument data stream) experiment used the input stream MSFT, one thread, and was measured for 38 loop runs (see Fig. 7). The graph shows the evolution of overtime with the number of loops runs executed. The full set of measurements for this experiment is given in the Table III.

Processing remains within real-time for 26 seconds and then takes a very steep evolution to go up to 4.5 seconds behind real-time. We attribute this sharp effect to Java garbage-collection marked *GC!* in the table, as it is visibly not the effect of an accumulating (time) delay, rather the effect of accumulating space in the thread's work space.

A defensive approach to this phenomenon would be to measure the accumulating memory consumption, allowing a-priori prediction of the garbage-collection start time. However, this would only allow the system to preventively stop this thread before it damages overall performance, not to prevent the end of its' real-time execution. A more aggressive approach would be for our system to generate code in a language where memory management is explicit (C, C++, Ada or OCaml with mutable variables) so that the memory leak is avoided.

The 2-symbol experiment used two input streams, instruments MSFT and INTC, two independent threads to process them, parameters T_{acc}, n and overtime were tabulated for each one up to 30 seconds. Because of the unpredictable behaviour of thread scheduling, it is not practical to predict/observe reliably which threads will be slowest to arrive at each loop cycle. In this case we compute overtime as formula $T_{\text{acc}} - (1000\text{ms}) \times n$ for the slowest thread at each loop.

TABLE I
PERFORMANCE DATA FOR THE 1-INSTRUMENT EXPERIMENT

Symbol	Exec(ms)	Nb. loops	Over-time(ms)
MSFT	1866	1	-134
MSFT	2857	2	-143
MSFT	3913	3	-87
MSFT	4902	4	-98
MSFT	5904	5	-96
MSFT	6908	6	-92
MSFT	7902	7	-98
...	...	...	...
MSFT	23906	23	-94
MSFT	24910	24	-90
MSFT	25905	25	-95
MSFT	26942	26	-58
MSFT	28039	27	39 (GC!)
MSFT	29041	28	41 (GC!)
MSFT	30216	29	216 (GC!)
MSFT	31215	30	215 (GC!)
MSFT	32349	31	349 (GC!)
MSFT	33382	32	382 (GC!)
MSFT	34394	33	394 (GC!)
MSFT	35392	34	392 (GC!)
MSFT	36381	35	381 (GC!)
MSFT	37415	36	415 (GC!)
MSFT	38409	37	409 (GC!)
MSFT	39450	38	450 (GC!)

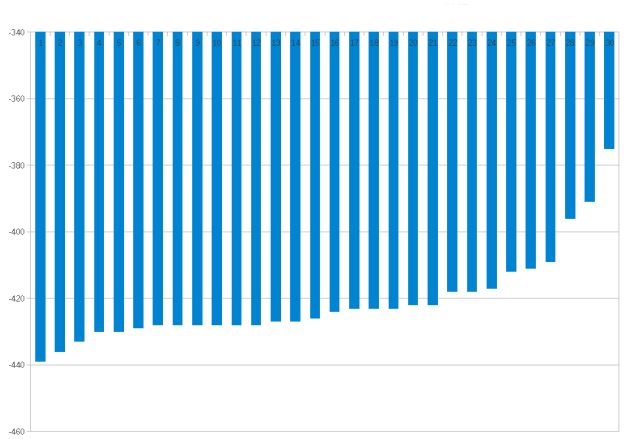

Fig. 8. Two-Symbol Experiment: Overtime vs Ticks.

The graph below shows the evolution of overtime with the number of loop runs executed.

With this setup, computation stays well within real-time by a margin of more than 300 ms, but the last few ticks show a degradation that might announce the garbage collection we had observed in the 1-symbol measurements.

The 4-symbol experiment used input streams for instruments GE, GOOG, INTC, MSFT, and also four independent streams to process them over 30 seconds. This time the overtime stayed at about twice that for two symbols and two threads: from -988ms, deteriorating slowly up to -971, and then a sharp deterioration during the last few seconds. The progress from one to two and four threads appear to show

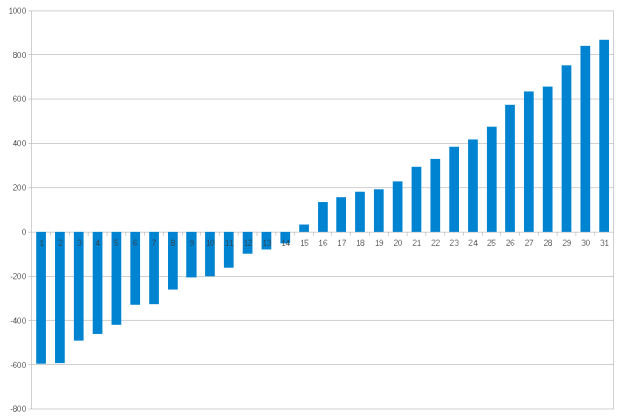

Fig. 9. Sixteen-Symbol Experiment: Overtime vs Ticks.

that overtime is (negatively) proportional to the number of streams and threads, but that is not the case. The 8-symbol experiment used input streams for instruments GE, DELL, SIRI, YHOO, GOOG, INTC, EBAY, MSFT and as many independent threads to process them. The overtime values were in fact very close to those for the 4-symbol experiment. Over 30 seconds they varied from -956ms to -155 very regularly. We explain this progression by the accumulation of delays that can be projected to non real-time processing after about 40 seconds in this case.

The 16-symbol experiment used input streams for instruments GE, INTC, YHOO, NVDA, ZNGA, EBAY, MU, ORCL, GOOG, MSFT, RIMM, DELL, RY, BNS, CSCO, SIRI, and sixteen independent streams to process them over 30 seconds. The evolution of overtime is displayed in the graph below.

The progress is clearly linear and we explain it by the accumulation of processing delays at every loop. A subset of the full measurement data for this experiment is shown in the Table II.

Similar observations were made with 10, 12 and 14 instruments: when the two hardware cores are sufficiently loaded, overtime evolves linearly with time. However, the initial level of overtime is not easily predicted from the number of symbol and threads.

TABLE II
PARTIAL PERFORMANCE DATA FOR THE 16-INSTRUMENT EXPERIMENT. "MAX" IS THE MAXIMUM OVER-TIME OVER THE 16 INSTRUMENTS (SYMBOLS).

symbol	Exec(ms)	N. loops	Over-time	Max.
GE	778	1	-1222	-595
INTC	1233	1	-767	-
YHOO	1234	1	-766	-
NVDA	1248	1	-752	-
ZNGA	1249	1	-751	-
EBAY	1267	1	-733	-
MU	1301	1	-699	-
ORCL	1323	1	-677	-
GOOG	1336	1	-664	-
MSFT	1341	1	-659	-
RIMM	1343	1	-657	-
DELL	1343	1	-657	-
RY	1369	1	-631	-
BNS	1371	1	-629	-
CSCO	1404	1	-596	-
SIRI	1405	1	-595	-
GE	2034	2	-966	- 593
INTC	2226	2	-774	-
YHOO	2233	2	-767	-
ZNGA	2253	2	-747	-
MU	2297	2	-703	-
ORCL	2318	2	-682	-
NVDA	2321	2	-679	-
GOOG	2329	2	-671	-
MSFT	2341	2	-659	-
DELL	2355	2	-645	-
RIMM	2364	2	-636	-
BAY	2375	2	-625	-
BNS	2387	2	-613	-
...	...	...	...	...
ORCL	31603	30	603	841
RY	31615	30	615	-
DELL	31647	30	647	-
MU	31660	30	660	-
ZNGA	31661	30	661	-
GE	31688	30	688	-
BNS	31714	30	714	-
RIMM	31715	30	715	-
SIRI	31722	30	722	-
GOOG	31723	30	723	-
INTC	31723	30	723	-
YHOO	31745	30	745	-
EBAY	31746	30	746	-
NVDA	31777	30	777	-
MSFT	31809	30	809	-
CSCO	31841	30	841	-
...	...	...	...	...

C. Scaling up the hardware and number of streams

Let us now outline solutions for scaling up the experiments so that more threads can be processed, and (positive) overtime delayed for as long as possible, then ultimately avoided.

Let us assume that garbage collection can be avoided by explicit memory management in the target language of our generated code.

Assume also a fair thread scheduler or, more natural for parallel processing, *processes* in place of threads. This will result in the elimination of variability between the various thread processing loops; in normal situations each one should run at the same speed, hence avoiding load-balancing problems even on a distributed-memory architecture.

If our 16-symbol experiment was run on a 4, 6, or 8-core system, the portion of timing responsible for the accumulated delay would be divided, respectively, by 2, 3, or 4. This would lead to a linear (in fact affine) decrease with the number of cores, a trend that would be visible in the initial parts of those experiments. If a large architecture is benchmarked in this manner, it becomes a simple matter to extrapolate the sufficient number of cores to process our streams in guaranteed real-time. Since those measurements are quickly and easily

performed within a few minutes (or in advance) they can become part of the automatic code generation process and thus ensure real-time processing for a known duration. This type of optimization can be completely automated and even hidden from the user if enough hardware resources are available.

D. Application to parallel portfolio analysis

Financial risk management and some trading strategies require real-time computations that process a whole portfolio of symbols at each input-decision loop. In pure computational terms this amounts to a situation similar to the one we have experimented with so far, except that a single trading strategy program inputs data from all streams in the portfolio and outputs related trading orders repeatedly. If the number of input streams is sufficiently large like in our 8- to 16-symbol experiments, then it is possible to structure a strategy as a parallel program whose processes input (subsets) of stream values, compute locally, communicate and arrive at a global trading decision based on global statistics.

Such a loop body is typical of data-parallel computation and our recent work on the SGL model [1], [2] has shown how to benchmark and extrapolate performance for such programs. SGL is a variant of Valiant's Bulk-synchronous parallel processing (BSP) and a generalization of Map-Reduce. It can deal with hierarchical and heterogeneous architectures that are now very common. It assumes algorithm regularities like those we have observed here, benchmarks the architecture's loop-processing speed, network latency and network bandwidth. From those parameters and the main data size factors it becomes possible to extrapolate performance to very large, even unavailable architectures in a reliable way. We therefore propose that the automatic code generation scheme presented in this paper can be adapted with an SGL-like performance model and data-parallel code generation, coupled with automatic adaptation to given performance constraints.

An early example of this concept is shown in graph Fig. 10 where we have experimented in April-May 2014 with an increasing number of threads and symbols (for 2 -, 4-, 8-, 16- and 23-symbol experiments) on Windows 2008 R2 VMWare Guest with 2 Intel Xeon CPUs (4 cores each). The curve's minimum is what our code-generation system can target for its chosen number of threads when time has to be minimized. The curve can also serve as performance prediction when the user wishes to set an arbitrary, non-optimal number of threads.

IV. Testing Results with BSP parallel processing

The performance experiments we have conducted 4 consisted in processing streams of real-time financial data from Yahoo finance (dated September 2014) taken from Yahoo as quickly as the hardware and software on either end would allow. Each input stream is a sequence of quotation values for one stock "symbol" among the following: GOOG, IBM, BBRY, DEL, APPL, CBL, CERE, CERN, CBOE, IGR, CBG, CAW.

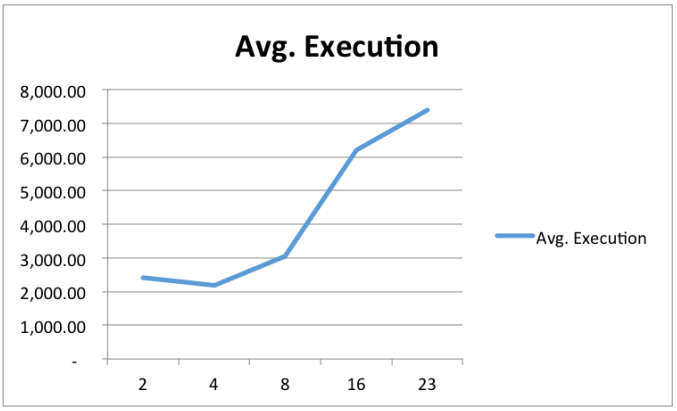

Fig. 10. Experimental Results with an Increasing Number of Threads.

A. The implementation of multi-threading

Parallelization was implemented by the multi-threading module with Bulk Synchronous Parallel (BSP) algorithm. After program initialization where the stock list is populated and the child threads are spawned we implemented an algorithm for the parallel section to use it's own id to determine what stocks it needed to grab to divide the number of stocks between the different threads. When the individual threads complete they inform the parent which can then continue and create the final output for the program (Fig. 11).

Program Processes:

HPCMulti

The range of the number of threads is from 1 to 8 (hard coded just for the experiment only). The range of stocks to load is from 4 to 24. The program will then print out basic information stating that all the data collected and placed into a new file called output.log for the statistical analysis. The program goes ahead and creates located space for timestamps and stocks determined by the number of threads the user wishes to use. The program then executes the implementation of JavaBSP from the class StockProcess.

Main Thread

The StockProcess class creates each thread determined by the number of threads from the initial user input from the command line. Splitting work between each of the threads. This then executes BSP_SYNC() allowing us to wait until the parallel threads are complete with their work.

Parallel Threads

Each thread will create a new WebParser connection to the Yahoo Finance API provided by Yahoo. The thread then creates a storage of Strings from the list of pre-initialized stocks in the equation (totalStocks / totalThreads) * (currentThread) to (totalStocks / totalThreads) * (currentThread + 1). Total stocks is the initial argument from the user. Total threads is the initial argument from the user. Current thread is labeled by each thread starting from zero. This will give the stocks

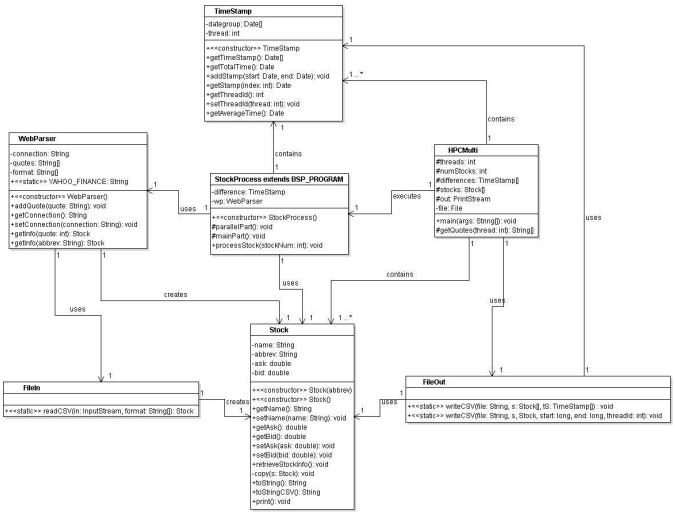

Fig. 11. Class Diagram of the BSP implementation.

from a list of stocks divided along each thread. If the number of total stocks is not evenly divisible by the total number of threads, the results are unpredictable. When the results are unpredictable, the deviation information maybe effected.

Once the stocks have been loaded, the process precedes to go through each stock and process it. Each processing of a stock will happen 10 times just to collect statistical information in our experimental setup, following the procedure: date, getInfo, date, create timestamp. The dates specify the start date of the process and the end date of the process. The getInfo represents retrieving the information from WebParser's URLConnection. Once the information is processed the thread will create a TimeStamp object to hold the millisecond difference it took to process the stock. The thread then inserts the stocks into a static variable in HPCMulti with the equation: ((currentThread / totalThreads) * totalStocks) + currentStock. Current thread is the current thread number. Total threads is the total number of threads the user specified. Total stocks is the total number of stocks the user specified. Current stock is the numerical value of the current stock processed in this thread.

The thread precedes to output information to output.log with the following information: Stock information and total time taken for that stock.

The thread then completes and closes.

WebParser

The WebParser initializes with a built in default format for processing Stocks: Name, Bid price and Ask price. The object then waits for information to fill it's list of Stock abbreviations, example: "GOOG", "IBM" and "BBRY. The information described are quotes, defined in the Yahoo exchange API. The connection is set as simple as webParser.setConnection(WebParser.YAHOO_FINANCE);. When the WebParser is asked to parse a Stock, it requires which stock to be processed, listing each quote from zero to the number of total quotes minus one. The getInfo() then connects to the Yahoo Finance API grabbing a CSV file and parses it into a Stock object using the FileIn class.

An automated trading strategy will generally process incoming data one at a time, accumulate some statistics, and issue orders BUY, SELL, or NIL. For each stock, we sent 10 requests for its stock information NAME, ASK PRICE and BID PRICE. The time for each request, the average for each group of requests, as well as the total time to gather the data across all threads is output and used to derive comparisons between different amounts of threads.

The generated Java code was run on a AMD Athlon II X4 640 3.00GHz processor, with 8GB ram, on 64 bit Windows 8.1

B. Performance data and analysis

In this section we have the results of our various experiments at various thread.

1-thread experiment using 4 symbols. The time for each individual request is given in Table III. The average of the stock groups and the total time for the entire experiment is given in Table IV.

The average request time is 100ms, and in a hypothetical average case the total time to do one request for each stock would take 400ms. There are a number of spikes in the requests times, the highest being 235ms, but as hoped these outliers are smoothed over by the preponderance of other data.

The 2-thread experiment, tables V and VI, used the same 4 symbols as the 1-thread experiment and the overall request average was higher, but as expected the total time it took for all the requests to be executed went down. The difference between 1 and 2 threads total times is not a perect 2:1 ratio, but this can be explained by the additional overhead required for spawning a new thread and the higher average request time.

The 4-thread experiment, VII used the same 4 symbols as the 1 and 2 thread experiments and the overall request average was lower this time, but as expected the total time it took for all the requests to be executed went down. The difference between 1 and 4 threads total times worked out to a perfect 4:1 ratio, the additional overhead created by spawning new threads offset by the lower average request time.

The experiment was run at the following symbol and thread amounts:

We gathered the results of the experiment and put it into the following graph Fig. 12.

The graphs represent the total time passed processing the data versus the number of stocks processed. The Y-axis is the total time that was taken to process the information in Milliseconds. The X-axis are the number of Stocks that were processed. The legend indicates the number of threads that the program was run with. The standard deviation points are determined by the program being processed multiple times providing the deviation result.

The specific deviation locations at 2 and 4 threads at stock 6 and 18 are not shown because of the testing environment not including values where stocks need to be divided evenly

TABLE III
PERFORMANCE DATA FOR THE 1-THREAD 4-SYMBOL EXPERIMENT

Symbol	Thread ID	Exec(ms)
GOOG	0	235
GOOG	0	140
GOOG	0	215
GOOG	0	100
GOOG	0	105
GOOG	0	100
GOOG	0	100
GOOG	0	100
GOOG	0	110
GOOG	0	100
IBM	0	100
IBM	0	100
IBM	0	100
IBM	0	100
IBM	0	100
IBM	0	100
IBM	0	100
IBM	0	110
IBM	0	100
IBM	0	100
BBRY	0	100
BBRY	0	100
BBRY	0	100
BBRY	0	100
BBRY	0	105
BBRY	0	105
BBRY	0	105
BBRY	0	95
BBRY	0	105
BBRY	0	100
DEL	0	105
DEL	0	100
DEL	0	100
DEL	0	110
DEL	0	100
DEL	0	105
DEL	0	100
DEL	0	105
DEL	0	100
DEL	0	100

TABLE IV
AVERAGE DATA AND TOTAL TIME FOR THE 1-THREAD 4-SYMBOL EXPERIMENT

Symbol	Thread ID	Average Exec(ms)
GOOG	0	100
IBM	0	100
BBRY	0	100
DEL	0	100
Total:400ms	Average: 100ms	

TABLE V
PERFORMANCE DATA FOR THE 2-THREAD 4-SYMBOL EXPERIMENT

Symbol	Thread ID	Exec(ms)
GOOG	0	200
GOOG	0	95
GOOG	0	100
GOOG	0	100
BBRY	1	565
GOOG	0	100
BBRY	1	90
GOOG	0	100
BBRY	1	95
GOOG	0	100
BBRY	1	90
GOOG	0	100
BBRY	1	110
GOOG	0	100
BBRY	1	95
GOOG	0	105
BBRY	1	90
BBRY	1	90
IBM	0	105
BBRY	1	90
IBM	0	100
BBRY	1	90
IBM	0	105
DEL	1	85
IBM	0	100
DEL	1	90
IBM	0	110
DEL	1	100
IBM	0	100
DEL	1	90
IBM	0	100
DEL	1	85
DEL	1	85
IBM	0	100
DEL	1	90
IBM	0	100
DEL	1	165
IBM	0	145
DEL	1	100
DEL	1	100

TABLE VI
AVERAGE DATA AND TOTAL TIME FOR THE 2-THREAD 4-SYMBOL EXPERIMENT

Symbol	Thread ID	Average Exec(ms)
GOOG	0	105
IBM	0	145
BBRY	1	90
DEL	1	100
Total:250ms	Average:110ms	

TABLE VII
AVERAGE DATA AND TOTAL TIME FOR THE 4-THREAD 4-SYMBOL EXPERIMENT

Symbol	Thread ID	Average Exec(ms)
GOOG	0	100
IBM	1	90
BBRY	2	90
DEL	3	90
Total:100ms	Average:92ms	

TABLE VIII
SYMBOLS AND THREADS

Threads	Symbols
1	4
1	6
1	8
1	12
1	16
1	18
1	24
2	4
2	8
2	12
2	16
2	24
3	6
3	12
3	18
3	24
4	4
4	8
4	12
4	16
4	24

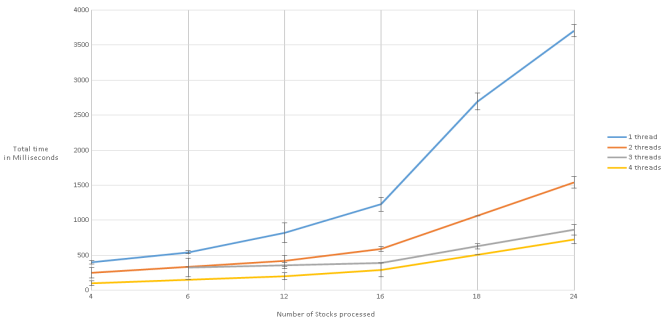

Fig. 12. Time Vs Threads

into the number of total threads. This applies with 3 threads at stock 4 and 16. As can be seen by the graphs as the number of threads increases, the total time to process information decreases.

As Fig. 12 shows, this increase is not linear but appears approximately as a quadratic growth in terms of the number of stocks processed for a given, fixed number of threads.

More precisely there are intervals where this relationship is linear but there is an upward inflexion around 16 stocks, irrespective of the number of threads.

In the dimension of processing scalability, we observe that the increment from 1 to 2, 3 and 4 threads for a fixed number of stocks almost always produces a non-negligible speedup.

But this speedup is very variable, from nil (8 stocks, 2 to 3 threads) to super-linear (24 stocks, 1 to 4 threads) and not easily predictable from our current experiments.

As a result, our ultimate goal of giving cost-vs-speedup predictions to users of our system will still require more experiments and analysis. Nevertheless, the experiments presented here clearly show that overheads can be non-negligible despite the apparently trivial nature of this parallel processing scheme, but that very significant speedups occur already with a small number of threads and cores.

V. RELATED WORKS

In [13] the authors describe model-driven software engineering, which allows constructing software from abstractions that are more closely fitted to the problem domain and that better hide technical details of the solution space. They used code generation to produce executable code from these abstractions. They have successfully generated EJB code from a class model, and are looking to further evaluate the process with more case studies, as well as in a comparative study with other generation approaches [13]. We performed code generation, not from a class diagram, but from a much higher abstract level - a business UML model, which has just business related information. We combined the UML business related model (business classes only - Fig. 1) with a library of predeveloped Java functions and optimized them for parallel execution.

Many software users perform tasks that vary on a yearly, monthly, or even daily basis. They explain that users software needs are diverse, complex, and frequently changing. Professional software developers cannot directly meet all of these needs because of their limited domain knowledge and because their development processes are too slow. As a result, end-user programming (EUP) helps to solve this problem by enabling end users to create their own programs. They defined end-user programmers (EUP) as people who write programs, but not as their primary job function they write programs in support of achieving their main goal, which is something else, such as accounting, designing a web page, doing office work, scientific research, entertainment, or engineering [17]. In our research we tried to minimize programming work for the EUP by using model driven development and parallel coding implementation for multi-core systems, which are widely used now in Industry.

While users in the past were satisfied with relatively simple interactive models of computation, such as spreadsheets and other business applications, current users are now looking to automate custom data manipulations such as reformatting, reorganizing, simple calculations, or data cleaning. While such users may have a good command of the interactive functionality of their application, they often lack the expertise, time, or inclination to develop software specifically for their task. These solutions will (at least initially) focus on the automatic generation of relatively small, but still useful solutions to everyday problems [18]. We tried to give a tool for automatic data collection and analysis to the end-users of a hedge fund in Paris, which allows them to update internal code and application behaviour without any coding skills by using a UML business model and predefined Java functions only. Our first part of the project was finished successfully and our client was satisfied by the Prototype.

VI. CONCLUSION

We have provided here a case study for performance optimization and code generation from customized UML models, for the example of a real-time algorithmic trading system.

During the case study, an advanced coding environment was developed for providing a visual and declarative approach used in the development of algorithmic trading strategies. This visual and declarative approach prevents errors related to different tools, and greatly simplifies developer challenges arising from the need to focus on financial algorithms, parallel computing and coding, etc. The coding environment utilizes a visual and declarative approach to developing algorithmic trading strategies from financial specifications. A new library of mathematical functions for stock interactions and a code generating plug-in for Eclipse were developed and implemented.

Additionally, we have conducted performance experiments to determine exact and quantitative conditions under which the system can adapt to varying data and hardware parameters. From these experiments we have shown how the financial-data stream processing can scale to a variable number of incoming streams and/or a variable number of hardware processing elements. For each parameter, we describe a method whereby our automatic code generation can adapt a priori to optimize performance.

We also outlined the scheme's adaptation to distributed-memory parallel processing in the presence of multiple streams and multiple processing elements. The automatic code generation scheme presented in this paper can be adapted with an SGL-like performance model and data-parallel code generation, coupled with automatic adaptation to given performance constraints.

VII. Future Work

The Prototype is a part of the ongoing joint research and software engineering project between Laboratory of d'Algorithmique, Complexité et Logique Université Paris-Est Créteil, France and Okanagan College, Canada. UBC O students initially started the Prototype in 2012 in COSC 319 SW Engineering Project [15]. In this project, we are attempted to design and implement an end-user programming environment where end-user code generation is accomplished through interaction with customized business UML models, and a predefined library of functions with performance optimization for multicore systems.

Additionally, the high-performance computing solutions are to be implemented for algorithmic trading systems from high-level abstract models to low-level programming languages by implementing the Low Level Virtual Machine compiler infrastructure [5].

The current version of the Prototype was implemented for the Windows OS only, but our clients wish to use multi-platform implementation, especially based on Linux platforms with automating scaling for multi-CPU and multicore systems, what is going to be accomplished in our next releases.

Acknowledgements

Our thanks go to COSC 319 SW Engineering Project's Team Epik 2012, UBC O: Brandon Potter, Jesse Gaston, Andrew Jankovic, Sam Boateng and William Lee [6].

We would like to thank to student research project sponsors: IBM, Oracle, Microsoft, VMWare, Greycon Group for supporting us, and to Amazon.com for 5 educational grants in 2010 - 2015 and for support from Okanagan College as well.

References

[1] C. Li and G. Hains, A simple bridging model for high-performance computing. High Performance Computing and Simulation (HPCS), 2011 International Conference on pp.249-256. [Online]. Available: http://dx.doi.org/10.1109/HPCSim.2011.5999831 (2011, Jul.)

[2] C. Li and G. Hains, SGL: towards a bridging model for heterogeneous hierarchical platforms. Int. J. High Performance Computing and Networking, Vol. 7, No. 2, pp.139-151. [Online]. Available: http://dx.doi.org/10.1504/IJHPCN.2012.046371 (2012, Apr.)

[3] C. Li, Un modèle de transition logico-matérielle pour la simplification de la programmation parallèle. PhD thesis, École Doctorale Math & STIC, UniversiteṔaris-Est [Online]. Available: http://tel.archives-ouvertes.fr/tel-00952082 (2013, Jul.)

[4] Y. Khmelevsky, G. Hains and C. Li, Automatic code generation within student's software engineering projects. In Proceedings of the Seventeenth Western Canadian Conference on Computing Education (WCCCE '12). ACM, New York, NY, USA, 29-33. DOI=10.1145/2247569.2247578 [Online]. Available: http://doi.acm.org/10.1145/2247569.2247578 (2012, May)

[5] The LLVM Compiler Infrastructure. [Online]. Available: http://llvm.org/

[6] B. Potter, J. Gaston, A. Jankovic, S. Boateng, and W. Lee, Parallel Code Generation Plug-in XML to Java Plug-in Project Documentation. Team Epik. COSC 319 "SW Engineering Project" Final Project Report, Computer Science Department, University of British Columbia Okanagan, Kelowna, BC, Canada (2012, Apr.)

[7] Y. Khmelevsky, Research and teaching strategies integration at post-secondary programs. In Proceedings of the 16th Western Canadian Conference on Computing Education (WCCCE '11). ACM, New York, NY, USA, 57-60. [Online]. Available: http://doi.acm.org/10.1145/1989622.1989638 (2011, May)

[8] Y. Khmelevsky, Software development projects in academia. In Proceedings of the 14th Western Canadian Conference on Computing Education (Burnaby, British Columbia, Canada, May 01 - 02, 2009). R. Brouwer, D. Cukierman, and G. Tsiknis, Eds. WCCCE '09. ACM, New York, NY, 60-64. [Online]. Available: http://doi.acm.org/10.1145/1536274.1536292s (2009, May)

[9] Eclipse Modeling Framework Project (EMF) [Online]. Available: http://www.eclipse.org/modeling/emf/

[10] Mean reversion (finance). [Online]. Available: http://is.gd/1yHRhJ

[11] Pairs trade. [Online]. Available: http://en.wikipedia.org/wiki/Pairs_trade

[12] G. Canfora, M. Di Penta and L. Cerulo, Achievements and challenges in software reverse engineering. Commun. ACM 54, 4, 142-151. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924451 (2011, Apr.)

[13] S. Zschaler and A. Rashid, Towards modular code generators using symmetric language-aware aspects. In Proceedings of the 1st International Workshop on Free Composition (FREECO '11). ACM, New York, NY, USA, Article 6 , 5 pages. [Online]. Available: http://doi.acm.org/10.1145/2068776.2068782 (2011, Jul.)

[14] F. Mischkalla, D. He and W. Mueller, Closing the gap between UML-based modelling, simulation and synthesis of combined HW/software systems. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1201-1206 (2010, Mar.)

[15] B. Potter, J. Gaston, A. Jankovic, S. Boateng, and W. Lee, Parallel Code Generation Plug-in XML to Java Plug-in Project Documentation: Source Code and Design Diagrams. Team Epik. COSC 319 "SW Engineering Project" Final Project Report, Computer Science Department, University of British Columbia Okanagan, Kelowna, BC, Canada. [Online]. Available: http://sourceforge.net/projects/pestplugin/files/ (2012, Apr.)

[16] C. Scaffidi, J. Brandt, M. Burnett, A. Dove, and B. Myers, SIG: End-user programming. In Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts (CHI EA '12). ACM, New York, NY, USA, 1193-1996. DOI=10.1145/2212360.2212421 (2012)

[17] H. Lieberman, F. Paterno, M. Klann, and V. Wulf, End-user development: An emerging paradigm, Springer (2006)

[18] M. C. Rinard, Example-driven program synthesis for end-user programming: technical perspective. Commun. ACM 55, 8 , 96-96. DOI=10.1145/2240236.2240259 (2012, Aug.)