Code generation and parallel code execution from business UML models: A case study for an algorithmic trading system

Gaetan J.D.R. Hains, Chong Li, Daniel Atkinson, Jarrod Redly, Nicholas Wilkinson, Youry Khmelevsky

Code Generation and Parallel Code Execution from Business UML Models: A Case Study for an Algorithmic Trading System

Gaétan Hains*, Chong Li†
LACL, Université Paris-Est Créteil, Paris, France
Emails: gaetan.hains@huawei.com, research@chong.li
*Also affiliated with Huawei France R&D Centre, Paris.
†Also affil. with National Inst. of Informatics, Tokyo, Japan.

Daniel Atkinson, Jarrod Redly, Nicholas Wilkinson and Youry Khmelevsky
Computer Science, Okanagan College, Kelowna, BC V1Y4X8
Emails: daniel_atkinson@mail.com, jarrod_redly@hotmail.com, {kharnaxin, khmelevsky}@gmail.com

***Abstract*—In this paper we discuss several capstone student projects conducted by the students at University of British Columbia, Okanagan campus (UBCO) and at Okanagan College in different years. The aim of the projects was to demonstrate how end-users could update code for an industrial application (an algorithmic trading system) without any programming skills and programming experience. Another goal was to improve performance for the applications collection of stock information from online public sources by introducing parallel code execution on multi-core personal computers. Real algorithmic trading system requirements were used as a case study. An Eclipse Modelling Framework was used to generate Java code from a UML business model, which can be modified by unexperienced business users. Moreover, code execution can be scaled to a specific computer architecture and hardware for better performance and better computer resources utilization, especially if a business user wants to collect and analyze a long list of stocks. The last section of the paper focuses on performance optimization and analysis.**

Keywords-UML; code generation; high performance computing; BSP; performance prediction; parallel programming; Algorithmic Trading

I. Introduction

”Algorithmic trading systems are widely used by many financial institutions like hedge funds, pension funds, mutual funds, market makers and other institutional or individual traders to manage market impact, risk, and to provide liquidity to the market. A professional trader usually diversifies 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. The quantitative analysts (Quant) tends to work in pairs with software engineers for coupling trading algorithms design, development and deployment. Quant handles the financial aspect and software engineer deals with computer performance and code quality” [5].

Li and Hains in [12], [11] proposed the SGL software-hardware bridging models implementation for the parallel code execution in software applications. In our research we tried to optimize end-user code by generating code for different platforms from customized UML models.

Okanagan College incorporated industrial projects into capstone COSC 224 ”Projects in Computer Science” and

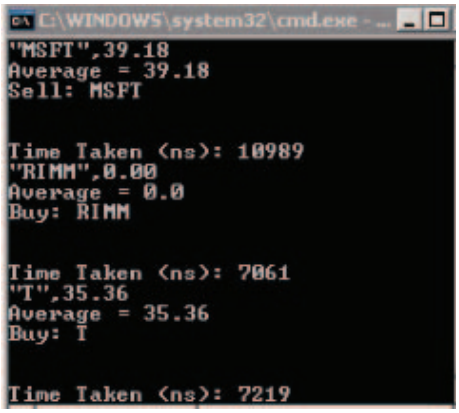

Fig. 2. Console output.

COSC 470/471 ”SW Engineering” project courses starting from 2006 [8], [7]. The discussed projects was originally started at UBCO in a software engineering course in 2012 [9], [15], [16], then continued at Okanagan College (OC), Kelowna campus in 2014 [6]. Both projects were supervised by a professor and his PhD student from Université Paris-Est, France [6].

In the first project UBCO students developed a code generating application by using Eclipse Modelling Framework (EMF) [4], which was able to successfully generate code from customized business models Fig. 1 and then run application in both modes: in console mode Fig. 2 and as a graphical application Fig. 3. The code generation was done via plugin implementation in Eclipse Modelling Framework (Fig. 4) by using .ecore XML files from UML models (Fig. 5). The source data were collected from Yahoo in real-time [15], [16].

Our contributions are the following:

1) A visual and declarative technique for the automatic Java code generation from UML business models in Eclipse Modelling Framework by the end-users without programming skills and experience.

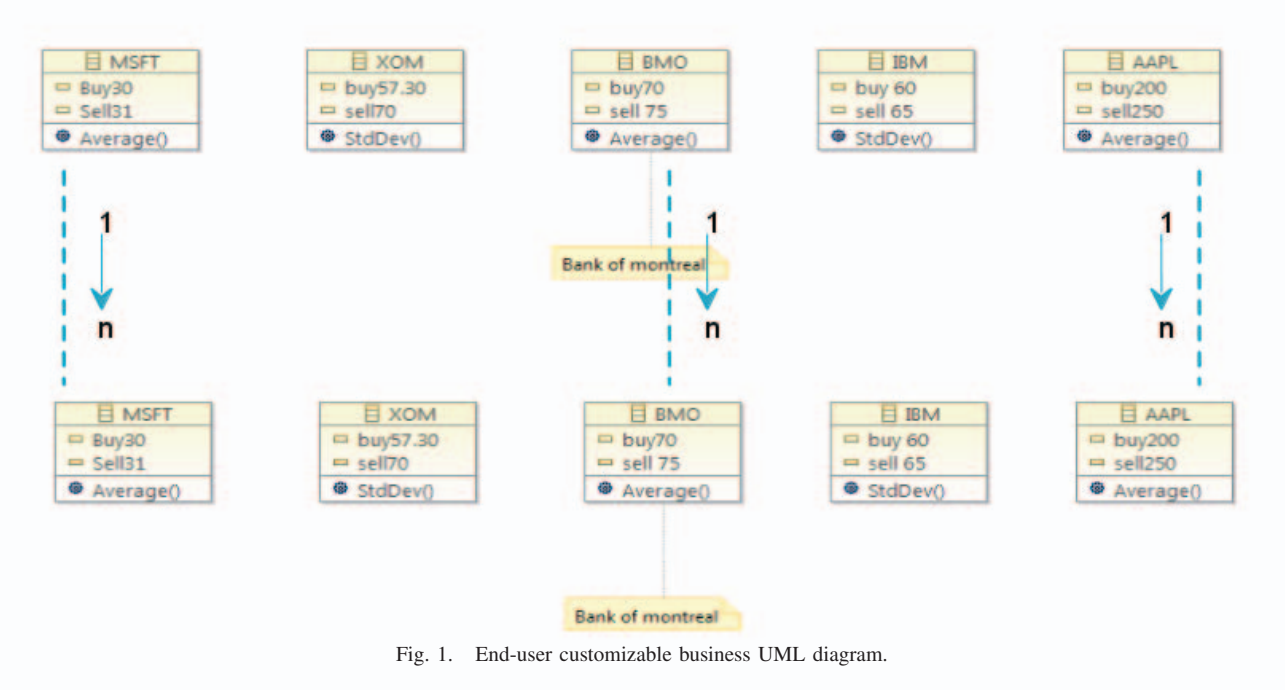

Fig. 1. End-user customizable business UML diagram.

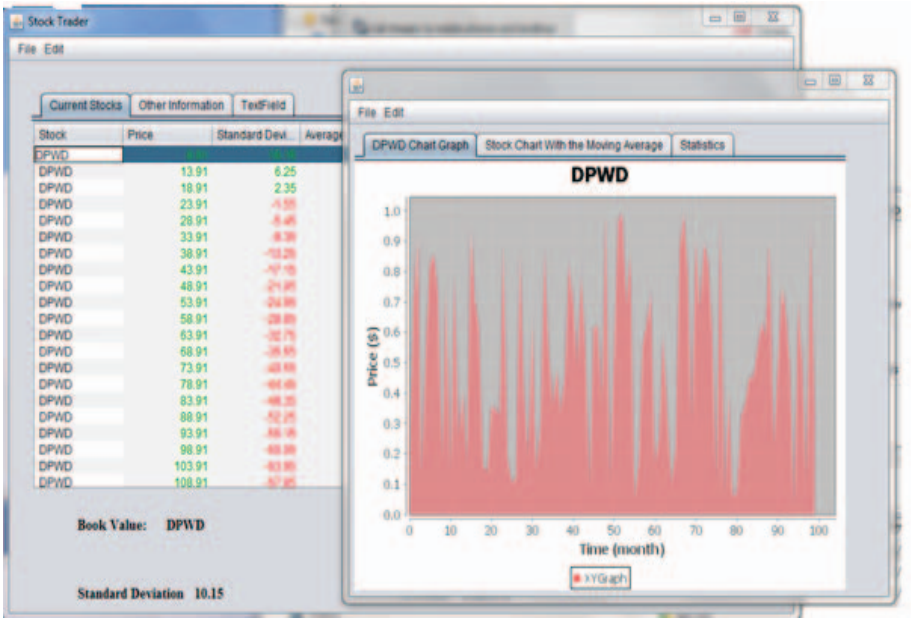

Fig. 3. Real-time application for the stock monitoring.

2) A software application prototype for a company in Paris.

3) Testing results of the BSP parallel code execution optimized for a multi-core personal computer. This can be adopted for different hardware platforms with automatic code regeneration.

II. Testing results with BSP parallel processing

To test the performance benefits of implementing multi-threading into the algorithmic trading system prototype, experiments were run using variable numbers of stocks and threads. Each experiment took streams of financial data from Yahoo Finance (in 2014 - 2015) in real-time at a rate of one record per second from a pre-initialized list of stock symbols. Each performance experiment was run at least 11 times in order to obtain 10 artifact free result sets with which to analyze the system's operation. In this section we discuss our latest original results, what we achieved by repeating all experiments on different hardware. The initial experiments with the BSP implementation were done in the Fall 2014 during COSC 470 capstone project.

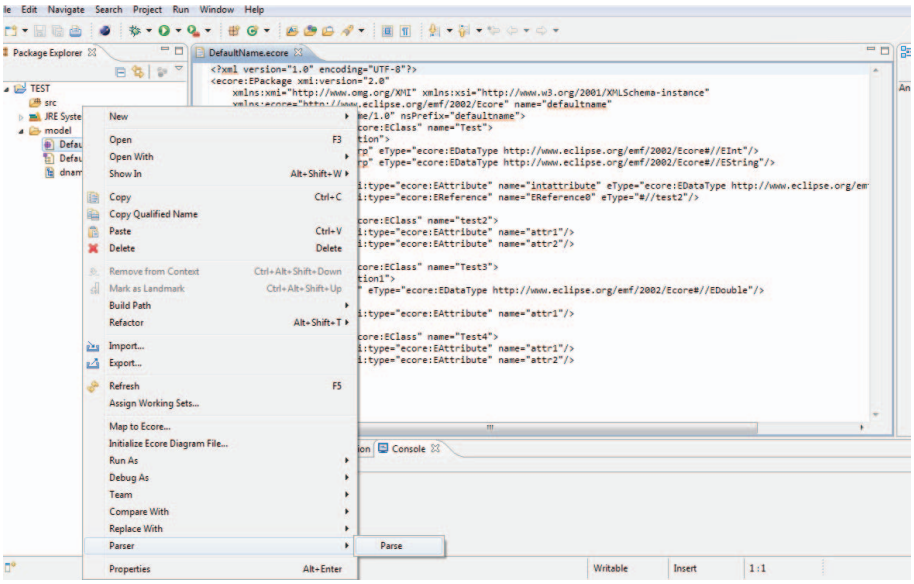

Fig. 4. The code generation plugin.

Fig. 5. .ecore XML file.

A. The Implementation of Multi-Threading

Taking the initial ”Algorithmic Trading System” prototype, we sought to integrate multi-threading techniques to improve performance and keep the application processing time in real-time in order to keep up with received market data. To achieve parallelization, the Bulk Synchronous Parallel (BSP) [19] algorithm was utilized for the multi-threading aspects of the program. A BSP model consists of a processor on which local computation is done, a communication network between these processors, and a synchronization barrier. To facilitate this model the MulticoreBSP Java framework [23] was integrated into the project which handled much of the model's implementation. The system informs the BSP framework of how many threads it needs and then the framework initializes the threads. Upon their creation the thread determines what stocks it needs to fetch which are divided between the created threads. After the thread finishes its execution then the thread calls the BSP sync method in the framework to inform the parent process of its completion. The parent after creating the threads will also have executed the sync method and will wait for all the child processes to synchronize, and once this occurs the parent can proceed to generate the output for the system (Fig. 7).

B. Program Processes

HPCMulti: This program handles the initial execution for the application. The user will begin by executing this file with parameters for the number of threads to use and

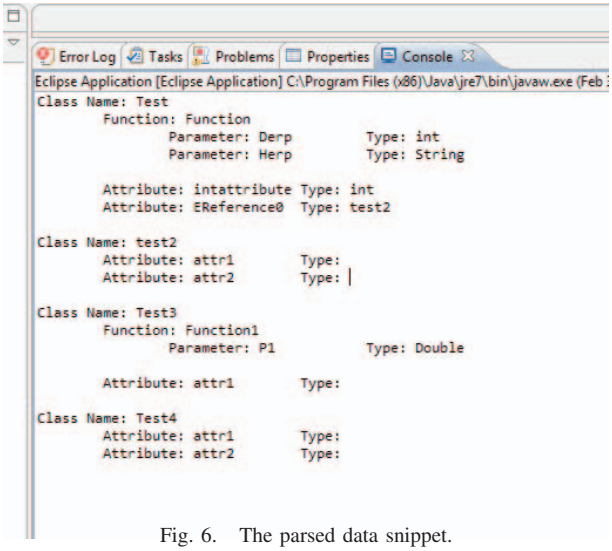

Fig. 6. The parsed data snippet.

the number of stocks to monitor. For the purposes of this experiment the number of cores that a user can select to utilize is limited to a maximum of either. Likewise, the range of stocks to load runs from a minimum of four up to a limit of 24. This program stores these values along with timestamps and a container for all the stocks. It also generates the output log into which is placed all statistical data for later analysis. The program then proceeds to execute the implementation of MulticoreBSP from the class StockProcess.

StockProcess: The StockProcess class handles the implementation of BSP for this program including the creation and management of the child processes which will collect the stock data, and the collation of this data into a CSV text file for statistical analysis. Discussion of this class is divided into two main sections; the main thread and the parallel threads. The main thread is the parent thread for all the child threads and handles the data output. The parallel threads create the individual connections to Yahoo Finance and collect data pertaining to their particular stocks.

The main thread manages the initialization of the child processes and then the printing out of the information that those threads gleam from the data source along with important statistical material regarding the execution time of the program. Using the variables that HPCMulti stored upon its execution, this method begins by initializing a number of threads equal to the number that the user requested at the command line by calling the BSP framework's begin function. This function will spawn the child processes for the program and start them executing the code contained in the parallel part. This then executes the BSP synchronize function which will force it to wait while the child processes execute their programming. When all threads have completed their operation and synchronized then this thread proceeds with its own execution. It writes out all stock data to a CSV file along with the time of execution and average execution time of each thread in milliseconds and informs the user of the newly created file.

The parallel section contains all code that each child process will run. Upon its creation by the BSP framework the thread will begin execution of this section starting with the creation of a WebParser object which will handle communication with the Yahoo Finance API. Continuing on, the thread fetches its selection of stocks from a pre-initialized list of stocks and stores them as an array of strings. The program divides the total number of stocks by the total number of threads to segment the list into a number of ranges and each process uses its own process id to find its range of stocks.

Once the thread has gathered its selection of stocks, it then proceeds to process each stock, one by one. For the purposes of this experiment the processing of a stock is repeated 10 times to collect and record statistical data regarding this execution. In addition to the actual stock data retrieved from the WebParser includes start and end date of the process and processing time. The generated Stock object is then inserted by the thread into a static variable in HPCMulti that contains all stock data for all retrieved stocks. The thread then injects the stock into a static array in HPCMulti that contains all stock data for all retrieved stocks.

When the thread has finished processing all of its stocks it logs the total time that was required by the thread to process all of its stocks in a static TimeStamp variable in HPCMulti. The thread then completes and closes.

WebParser: The WebParser handles the fetching of stock data from Yahoo Finance and parsing that into a Stock object that the application will read. On initialization the WebParser contains a list for stock abbreviations and a default format for processing stocks. The list is empty at initialization and the object awaits abbreviations to be passed to it to be added to the list. The parser then waits until it is requested to parse a stock either in the form of a string stock abbreviation or the integer location of an abbreviation in its list. The object then connects to the Yahoo Finance API and, using the FileIn class, parses the data received as a CSV file into a Stock object containing its name, ask price, and bid price. An automated trading strategy will go through the data one stock at a time and interpret it by accumulating statistics and issuing BUY, SELL, or NIL orders.

C. Performance Data and Analysis

In this segment we will display and analyze the performance results of our experiments using various numbers of threads and stocks. All results were retrieved by running the program on an AMD Phenom II X6 1055T 2.80GHz CPU with 8GB of RAM on Windows 7 SP1 64-bit.

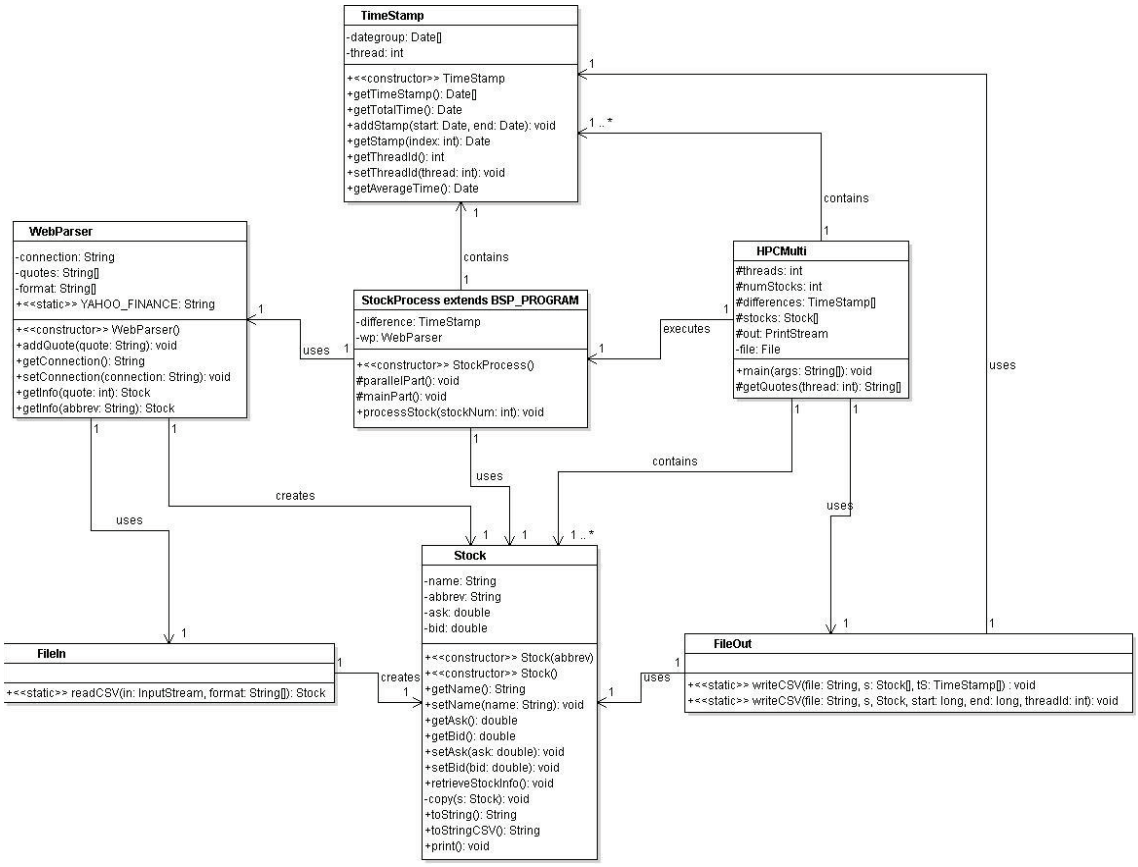

Fig. 7. Class Diagram of the BSP implementation.

TABLE I
1X4 EXPERIMENT (1 THREAD X 4 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	50	7.45
IBM	51	10.49
BBRY	50.8	12.24
DEL	48	7.89
Avg. Total:199.8ms	Avg. Exec:49.5ms	Std.Dev.:7.69

TABLE II
2X4 EXPERIMENT (2 THREADS X 4STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	48.2	5.07
IBM	48.4	5.87
BBRY	49.3	9.97
DEL	47.1	5.47
Avg. Total:98.9ms	Avg. Exec:47.6ms	Std.Dev.:5.06

Looking at the results for our experiments we start with experiments using four symbols. Each experiment was run at least 11 times to get 10 artifact free trials using the exact same set of four stocks with the only variation being the number of threads that the test employed. Starting with the 1-thread experiment using four symbols, the time for each test and the average time to process each stock is given in Table I. The average time for a stock to be processed was 49.5ms with a standard deviation of 7.69, and the average time to process all four stocks being 199.8ms. Due to the preponderance of the other data, the outlying data spikes in this set have been smoothed over granting a clear view of the content found within.

The experiment in Table II employed two threads and the same four stock symbols. Overall the average request time went down to 47.6ms, and as expected the average total time of to process all requests went down to 98.9ms. Comparing the average total times of one and two threads reveals an expected 2:1 ratio, because although the spawning of new threads incurred additional overhead by separating the four stocks onto two threads was able to offset this extra cost such that it reaches that 2:1 ratio.

The 4-thread experiment using the same four symbols, Table III, as the 1 and 2 thread experiments again had a lower average request time, down to 45.6ms, and as expected a drop in the total time, down to 48.8ms, it took for all the requests to be executed. Comparing the average total times for one and four threads results in an almost perfect 4:1 ratio

TABLE III
4X4 EXPERIMENT (4 THREADS X 4STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	45.8	4.69
IBM	45.3	4.14
BBRY	46.3	3.53
DEL	46.2	4.18
Avg. Total:48.8ms	Avg. Exec:45.6ms	Std.Dev.:3.06

TABLE IV
4 STOCKS EXPERIMENT

Threads	Avg. Total(ms)	Fastest(ms)	Slowest(ms)	Std.Dev.
1	199.8	160	250	31.4
2	98.9	86	123	11.37
4	48.8	45	56	3.43

with the lower average execution time and added efficiency of four processes offseting the added cost of additional threads. Likewise comparing the results of the two thread experiment to this one shows a 2:1 ratio.

Table IV gives a summary of all three experiments. This table reveals a dramatic decrease in execution times in all columns as the number of threads is increased. As previously noted, in these experiments, the increase in threads directly correlated to an exact ratio increase with 1 to 2 threads resulting in a 2:1 improvement in total execution time, and moving from 2 threads to 4 results again in a 2:1 improvement in total execution time. In fact when comparing the slowest 4-thread execution against the fastest 1-thread execution the resulting ratio works out to 2.85:1. The table also reveals that as the number of threads increase the deviation decreases.

The experiment was repeated with a variety of thread numbers with 12 symbols. Each experiment was run at least 11 times in order to get 10 artifact free trials using the exact same set of 12 stocks with the only difference being the number of processes that the test was able to use. First in Table V is an 1-thread experiment with 12 symbols. The average time for a stock to be processed was 45.3ms with a standard deviation of 2.41, and an average time to process all 12 stocks being 548.7ms, less than the 3:1 ratio that was expected as compared to the four symbol experiment though this can be explained by the lower average request time.

The 3-thread, 12-symbol experiment, Table VI, shows a higher average request time, up from 45.3ms in the 1-thread experiment to 47.1ms here. With an average total execution time of 200ms, the average total time of execution is lower than in the 1-thread experiment, as was expected; however, the difference between one and three threads does not mirror a 3:1 ratio, but this can be explained by an overall higher average request time and the additional overhead required to spawn the new threads.

TABLE V
1X12 EXPERIMENT (1 THREAD X 12 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	45	5.96
IBM	45.4	3.86
BBRY	45.2	3.19
DEL	46	3.02
APPL	44.1	3.9
CBL	43.9	4.7
CERE	46.1	7.29
CERN	48.4	11.26
CBOE	48.6	12.32
IGR	44.9	3.35
CBG	47	7.15
CAW	44.1	3.03
Avg. Total:548.7ms	Avg. Exec:45.3ms	Std.Dev.:2.41

TABLE VI
3X12 EXPERIMENT (3 THREADS X 12 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	47.2	2.48
IBM	46.9	7.26
BBRY	46.3	6.07
DEL	46.2	6.01
APPL	47.4	6.04
CBL	49	9.37
CERE	46.8	6.43
CERN	47.9	5.93
CBOE	53.6	9.95
IGR	45.9	7.84
CBG	46.1	3.21
CAW	49.5	5.87
Avg. Total:200ms	Avg. Exec:47.1ms	Std.Dev.:3.98

The 6-thread, 12-symbol experiment, Table VII, reveals a lower average request time when compared to the 1-thread experiment, down to 44.7ms, and, as expected, a lower average total execution time. Comparing the 99.1ms average total time of execution to the 1-thread reveals a ratio of close to 5.5:1 and not the expected 6:1. Despite the lower average request time, it was unable to offset the additional overhead necessary to spawn the extra threads used in this experiment.

Table VIII offers a summary of all three discussed experiments as well as experiments with two threads and four threads. Similar to the other experiments despite displaying dramatic reductions in average total execution time neither the two thread or four thread experiment reach ratios of 2:1 or 4:1 respectively. Worth noting is that the fastest 3, 4, and 6 thread experiments meet or exceed their expected ratios when compared against the hypothetical, average 1-thread process, with the fastest 6-thread experiment achieving a 6.38:1 ratio when compared to the average 1-thread process. Also of interest is that when comparing the average 6-thread execution against the average 3-thread execution, the result is an almost perfect 2:1 ratio showing that at these numbers it

TABLE VII
6X12 EXPERIMENT (6 THREADS X 12 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	46.4	7.03
IBM	41.6	4.25
BBRY	43.2	5.35
DEL	42.7	2.41
APPL	47.3	8.6
CBL	43.2	2.57
CERE	49.4	7.82
CERN	45.4	4.99
CBOE	45.9	9.11
IGR	44.9	3.25
CBG	49.5	12.01
CAW	44.9	2.47
Avg. Total:99.1ms	Avg. Exec:44.7ms	Std.Dev.:3.71

TABLE VIII
12 STOCKS EXPERIMENT

Threads	Average(ms)	Fastest(ms)	Slowest(ms)	Std.Dev.
1	548.7	505	610	31.57
2	287.5	263	370	31.13
3	200	185	245	19.01
4	146.2	135	155	6.41
6	99.1	86	119	10.57

is possible for the benefit of additional threads is capable of offsetting the additional overhead. Similar to the four symbol experiments there is a clear decrease in standard deviation as the number of threads increase with a slight uptick on the 6-thread experiment which is worth mentioning.

The experiment was also conducted with various numbers of threads with 24 symbols. Each experiment was run at least 11 times in order to get 10 artifact free trials using the exact same set of 24 stocks was used in all iterations of the 24 symbol experiment. First in Table IX is a 1-thread experiment with 24 symbols. The average request time was 47ms and the average total execution time was 1139.9ms. As compared with the 1-thread, 4-symbol experiment discussed above this experiment's average total time falls short of the expected 6:1 ratio, although this can be explained by the lower average request time.

The 3-thread, 24-symbol experiment, found in Table X, shows a lower average stock request time, 45.4ms, but the average total time went down to 386.3ms. The difference between the 1 and 3 thread experiments narrowly misses the 3:1 ratio. This phenomena can be explained by the overhead to spawn additional threads.

Table XI shows 6-thread by 24-symbol experiment with a lower average stock request time, 44.3ms, and a lower average total time, down to 262.2ms. Again the ratio of the results of the 1-thread experiment to the results of this experiment fall short of the expected 6:1 ratio.

TABLE IX
1X24 EXPERIMENT (1 THREADS X 24 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	49.5	8.41
IBM	47.2	5.39
BBRY	48.9	5.34
DEL	48.6	4.79
APPL	45.7	3.74
CBL	46	3.4
CERE	47.5	5.46
CERN	46.3	3.47
CBOE	48.5	8.28
IGR	46.8	5.53
CBG	49.9	7.65
CAW	46.1	3.38
CDI	46.2	3.26
CDKVV	45.9	3.75
CDW	45.8	3.58
CECE	45.8	4.05
FUN	45.7	3.02
CDR	47.9	8.31
GTU	49.2	6.48
CGI	47.1	4.33
CPXX	54.4	13.34
CLS	46.8	4.47
CELG	46.3	4.64
CELGZ	47.8	5.9
Avg. Total:1139.9ms	Avg. Exec:47ms	Std.Dev.:3.86

TABLE X
3X24 EXPERIMENT (3 THREADS X 24 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	45	4.71
IBM	44.1	3.98
BBRY	46	6.15
DEL	46.6	4.09
APPL	48	10.06
CBL	45.1	4.72
CERE	48	10.59
CERN	46	3.94
CBOE	46	3.94
IGR	49	7.38
CBG	47.5	3.54
CAW	47	4.22
CDI	46	3.94
CDKVV	47	6.32
CDW	46	3.16
CECE	45.8	4.44
FUN	45.5	3.69
CDR	43.5	3.37
GTU	46	5.16
CGI	46.6	5.38
CPXX	44.4	3.06
CLS	44.7	4.81
CELG	46.5	4.12
CELGZ	46.5	3.37
Avg. Total:386.3ms	Avg. Exec:45.4ms	Std.Dev.:1.9

TABLE XI
6X24 EXPERIMENT (6 THREADS X 24 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	44.7	2.95
IBM	47.2	3.08
BBRY	45	3.33
DEL	45	3.33
APPL	43.5	3.37
CBL	43.9	4.12
CERE	44.7	4.69
CERN	44.3	4.14
CBOE	43.5	2.42
IGR	44.4	2.84
CBG	45	4.71
CAW	44.2	4.52
CDI	45.5	2.84
CDKVV	44.9	4.72
CDW	47.5	8.25
CECE	45	3.33
FUN	48.7	12.26
CDR	44.4	5.5
GTU	46	6.99
CGI	43.7	3.97
CPXX	44.2	3.77
CLS	45	3.33
CELG	44.4	4.97
CELGZ	44	3.94
Avg. Total:197.6ms	Avg. Exec:44.3ms	Std.Dev.:1.06

TABLE XII
8X24 EXPERIMENT (8 THREADS X 24 STOCKS)

Stock	Average Execution	Standard Deviation
GOOG	47.1	9.5
IBM	45.6	4.48
BBRY	44.6	4.5
DEL	48.5	7.28
APPL	44.6	4.27
CBL	45.7	6.31
CERE	43	2.91
CERN	43.5	4.86
CBOE	43.9	3.81
IGR	45.6	4.4
CBG	43	3.68
CAW	43.8	4.34
CDI	48.9	7.02
CDKVV	43.5	4.99
CDW	44.8	5.41
CECE	47.8	8.59
FUN	48.4	8.49
CDR	45	4.11
GTU	44.8	6.16
CGI	45.8	5.18
CPXX	42.8	4.47
CLS	47.6	8
CELG	44.8	2.74
CELGZ	43.8	2.86
Avg. Total:149.5ms	Avg. Exec:44.6ms	Std.Dev.:2.17

The 8-thread experiment with 24 symbols displays an average stock request time of 44.6ms in Table XII, and, as expected, a lower average total time, down to 149.5ms. As with the 6-thread experiment the average total time does not match up to the expected 8:1 ratio as compared to the 1-thread experiment but rather ends up at 7.6:1. This can be explained by an additional overhead cost required in the spawning of new threads which offsets the lower average stock request time, 44.6ms here versus 47ms in the 1-thread experiment.

Table XIII gives a summary of all three discussed experiments as well as experiments with two threads and with four threads. Unlike the other experiments the 2-thread and 4-thread experiments were able to achieve an almost exact 2:1 and 4:1 ratio respectively, each able to offset the cost of spawning new threads with increased processing efficiency. This table shows that there are significant benefits to be had from implementing multi-threading in a system like this even if it does not reach the expectations that we had. It is of note that when comparing the fastest 2, 3, 4, 6, and 8 thread executions against the hypothetical 1-thread execution the result of each one meets or exceeds their expected ratio. As with the other two experiment groups the standard deviation decreases dramatically as the number of threads is increased.

All the data we collected from the experiments was taken and charted in the graph found in Fig. 8.

TABLE XIII
24 STOCKS EXPERIMENT

Threads	Average Execution	Min(ms)	Max(ms)	Standard Deviation
1	1139.9	980	1285	91.52
2	575	505	626	42.7
3	386.3	360	430	22.09
4	285.1	275	315	13.56
6	197.6	185	220	9.95
8	149.5	140	172	10.06

The graph displays the amount of time required to process stocks using various numbers of threads by the number of stocks that they processed. Measured on the Y-axis is the length of time in milliseconds necessary for the processing. The X-axis shows the total number of stocks. Each line shows number of executing threads. Additionally, the standard deviation was calculated by measuring 10 sets of results generated by the program for each experiment.

Shown on the graph are the experiments for 1, 2, 3, 6, and 8 threads. These experiments were chosen because they showed a noticeable difference from one to the next and so their values were easy to read. Experiments were however, conducted for 4, 5, and 7 threads.

Fig. 8 displays a linear increase in processing time for a number of stocks and threads, although analysis of the data suggests that this may not be the case especially as the number of stocks is increased with total execution time

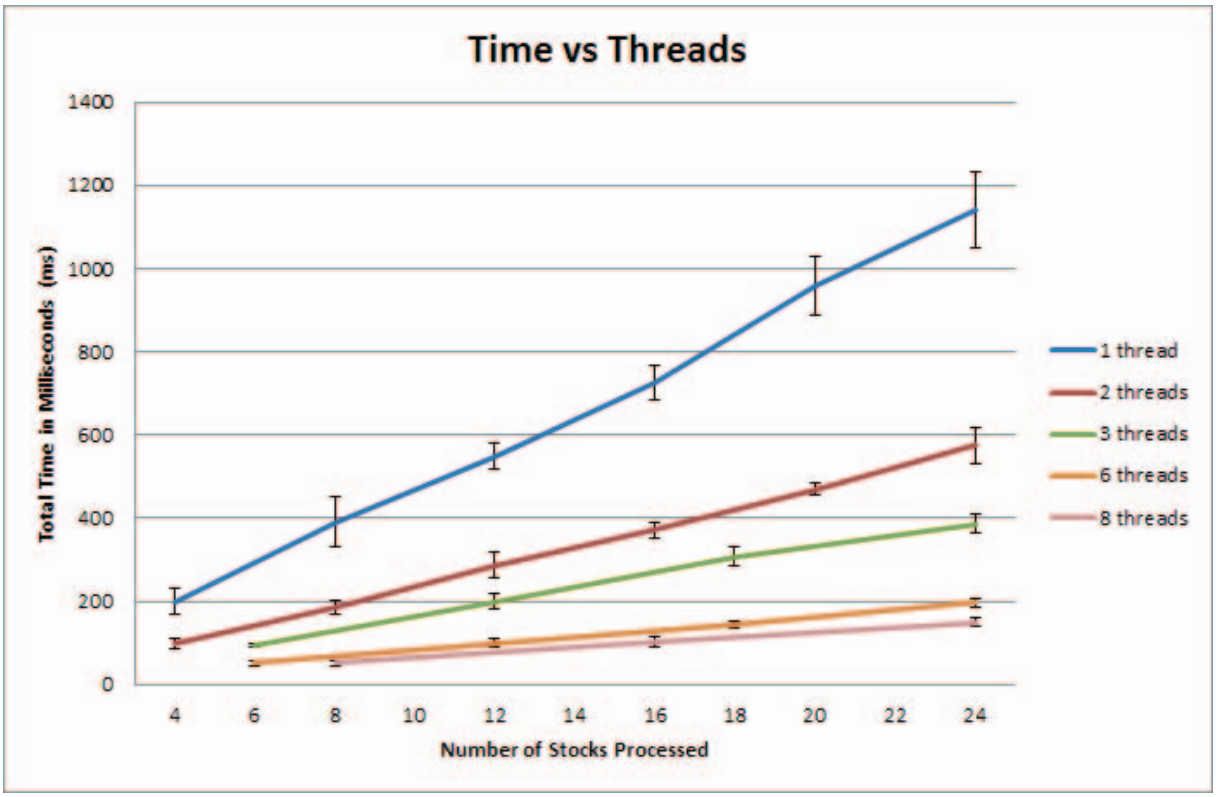

Fig. 8. Time vs threads

differences falling short of expected goals.

In terms of processing scalability, we observe that by increasing the number of threads almost always improve performance Fig. 8, performance improvement is non-linear for different number threads and stocks. But this improvement is different for different stocks and threads. Our goal of providing users with a cost-benefit prediction of this algorithmic trading system requires more experimentation and analysis.

Nevertheless, the experiments provided demonstrate that overhead costs for generating new threads can be non-negligible despite the ostensibly insignificance of their expense using this parallel processing scheme, but that impressive increases in speed can be obtained with a small number of cores and threads.

III. Related works

In [17] the author discusses a central question in computing related to computer programming and automation of this process, as well as about end user programming. ”Given the difficulty of specifying and implementing large software systems, these solutions will (at least initially) focus on the automatic generation of relatively small but still useful solutions to everyday problems”, what was actually done in our several student research projects. H. Liberman at el. in [13] explains, that ”the goal of human-computer interaction will evolve from just making systems easy to use (even though that goal has not yet been completely achieved) to making systems that are easy to develop”, but until now the development or modification of new applications requires considerable knowledge and programming expertise and skills, which is still rare for most people. In our paper an example was given for how a simple industrial application can be easily modified by the end-users without any deep experience in programming and software development and the generated application can be adopted to the specific hardware.

F. Mischkalla et el. in [14] has shown, that UML is used for the software design and modelling, but ”there is still a big gap from UML modelling to the code synthesis environment”. They demonstrates SystemC code generation from SysML UML in two case studies. In our case study we demonstrated how to use Java code generation from the widely used UML business models. On the other hand, C. Scaffidi et el. confirmed in [18], that ”as users continue to grow in number and diversity, end-user programming is playing an increasingly central role in shaping software to meet the broad, varied, rapidly changing needs of the world”. Many modern companies are concentrated on the development of software development tools ”enabling end users to create programs”. In [24] was confirmed, that ”code generation is used to produce executable code from abstraction models” and it’s ”closely fitted to the problem domain, and that better hide technical

details" of the programming solutions. In [3], [20] authors discuss different topics about code generation from UML models as well, including "simulation, model-checking or test generation tools". This is a growing area of interest, because it can reduce cost and increase accuracy, maintainability of the software applications. The "rapid development of high quality code is achievable by model-driven code generation" as well as a reduction in errors as compared to the manual code development [2].

IV. Conclusion

In this student research paper we discussed results of code generation from business UML models and testing results of parallel code execution for an Algorithmic Trading System industrial software prototype. Our testing results confirm that code execution by using BSP parallel model can improve performance of the generated application on multi-core systems. This is important nowadays because every year we have more cores in personal computers, but CPU speed has remained almost the same. That means we need to find better ways to adopt our industrial software applications to changing hardware. Redesign and reprogramming of computer applications is time consuming professional work, but in our case study we proved that code regeneration from UML business models can be done relatively easily even without any programming experience by the end-users. Additionally, code parallelization can be achieved and optimized for the current hardware platform by using code generation from UML models with using already developed libraries. That means, if the business user changes hardware, the application execution can be optimized automatically to the new hardware platform without any additional time consuming programming work.

In our performance experiments we sought to demonstrate how the system operates when given varying data and hardware specifications. The results obtained have shown that the system we designed scales to process the variable number of stocks and/or the variable number of CPU's cores with which to operate and that performance benefits are obtained. We displayed a selection of results for a variety of experiments that were run and discussed how they related to one another in terms of performance and posited reasons behind each outcome.

Acknowledgement

Our thank to the sponsors of the Okanagan College Gray-Con Group and Amazon, and to the University Paris-Est Créteil for their support of the student capstone projects.

References

[1] Computer Science Department at the University of Illinois at Urbana-Champaign. The llvm compiler infrastructure: `http://llvm.org`, 2015.

[2] Jeannette Bennett, Kendra Cooper, and Lirong Dai. Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach. *Sci. Comput. Program.*, 75(8):689–725, August 2010.

[3] Franck Chauvel and Jean-Marc Jézéquel. Code generation from uml models with semantic variation points. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 54–68, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Eclipse Foundation. Eclipse modeling framework (emf): `http://www.eclipse.org/modeling/emf/`, 2015.

[5] Gaétan Hains, Chong Li, Youry Khmelevsky, Brandon Potter, Jesse Gaston, Andrew Jankovic, Sam Boateng, and William Lee. Generating a real-time algorithmic trading system prototype from customized uml models (a case study). 2012.

[6] Gaetan Hains, Chong Li, Nicholas Wilkinson, Jarrod Redly, and Youry Khmelevsky. Performance analysis of the parallel code execution for an algorithmic trading system, generated from uml models by end users. In *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*, pages 1–10. IEEE, 2015.

[7] Youry Khmelevsky. Sw development projects in academia. In *Proceedings of the 14th Western Canadian Conference on Computing Education*, pages 60–64. ACM, 2009.

[8] Youry Khmelevsky. Research and teaching strategies integration at post-secondary programs. In *Proceedings of the 16th Western Canadian Conference on Computing Education*, pages 57–60. ACM, 2011.

[9] Youry Khmelevsky, Gaétan Hains, and Chong Li. Automatic code generation within student's software engineering projects. In *Proceedings of the Seventeenth Western Canadian Conference on Computing Education*, pages 29–33. ACM, 2012.

[10] Chong Li. *Un mode'le de transition logico-mate rielle pour la simplification de la programmation paralle'le.* PhD thesis, E cole Doctorale Math & STIC, Universite P aris-Est., July 2013.

[11] Chong Li and Gaétan Hains. A simple bridging model for high-performance computing. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 249–256. IEEE, 2011.

[12] Chong Li and Gaétan Hains. Sgl: towards a bridging model for heterogeneous hierarchical platforms. *International Journal of High Performance Computing and Networking*, 7(2):139–151, 2012.

[13] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006.

[14] Fabian Mischkalla, Da He, and Wolfgang Mueller. Closing the gap between uml-based modeling, simulation and synthesis of combined hw/sw systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1201–1206. IEEE, 2010.

[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.

[16] 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. April 2012.

[17] Martin C Rinard. Example-driven program synthesis for end-user programming: technical perspective. *Communications of the ACM*, 55(8):96–96, 2012.

[18] Christopher Scaffidi, Joel Brandt, Margaret Burnett, Andrew Dove, and Brad Myers. Sig: end-user programming. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 1193–1996. ACM, 2012.

[19] Utrecht University. The bulk synchronous parallel model `http://www.multicorebsp.com/main/model/`, 2013.

[20] Muhammad Usman, Aamer Nadeem, and Tai-hoon Kim. Ujector: A tool for executable code generation from uml models. In *Proceedings of the 2008 Advanced Software Engineering and Its Applications*, ASEA '08, pages 165–170, Washington, DC, USA, 2008. IEEE Computer Society.

[21] Wikipedia. Mean reversion (finance): `http://tinyurl.com/p9rlhfc`, 2015.

[22] Wikipedia. Pairs trade: `http://tinyurl.com/dkgrfa`, 2015.

[23] Albert-Jan N. Yzelman. Multicorebsp for java: `http://www.multicorebsp.com/download/java/`, 2011.

[24] Steffen Zschaler and Awais Rashid. Towards modular code generators using symmetric language-aware aspects. In *Proceedings of the 1st International Workshop on Free Composition*, page 6. ACM, 2011.