Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognize the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
TOWARDS OBJECT-ORIENTED OPENMP PARALLEL PROGRAMMING

XING FAN

Supervised by Dr. Nasser Giacaman and Dr. Oliver Sinnen

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer System Engineering
The University of Auckland, December 2017
Xing Fan: Towards object-oriented OpenMP parallel programming

© December 2017

SUPERVISORS:
Dr. Nasser Giacaman
Dr. Oliver Sinnen
ABSTRACT

With the coming of the data-driven era, software applications are becoming increasingly more computationally demanding. High-performance computing is more important than ever. Since Moore’s Law has reached its limit, parallelization is the key technique to reduce computational time. However, writing parallel programs is still notoriously difficult because it involves a great deal of expert knowledge, as well as programming and tuning efforts. Even though there are already many tools and languages addressing this challenge, it is still rare to see parallel techniques employed in the software industry.

OpenMP is a programming interface standard that facilitates writing multi-threaded applications. It adopts the fork-join model, allowing programs to be parallelized incrementally. By adding directives, a sequential program can be compiled to a parallel version without extra coding effort. The good usability of OpenMP makes it widely used in many practical parallel programs; however, it has been discovered that OpenMP is not closely incorporated with object-oriented programming concepts.

This thesis addresses that a good parallel programming model should be closely incorporated with software design principles, especially object-oriented programming concepts. This research identifies the gaps between object-oriented programming and OpenMP, and provides solutions to combine these two models in harmony.

The ultimate aim of this research is to help programmers write parallel code in a more efficient and less error-prone way. Throughout the thesis, Pyjama is demonstrated, which is a research project developed as an extended OpenMP parallel programming model for high-level object-oriented languages. The implementation is dedicated to Java.

The first important aspect contributed in this thesis is the proposal of enhanced exception handling within OpenMP, for object-oriented languages. The research identifies existing problems and proposes a robust exception handling mechanism to solve the
problems. The evaluation shows that the new approach provides an elegant and robust mechanism without causing any performance degradation.

The second significant contribution of this research is the development of an asynchronous model for event-driven programming. This model is regarded as compensation for the fork-join model, making Pyjama more powerful for event-driven programming patterns. New OpenMP directives are proposed to endow sequential code with an asynchronous execution nature. This idea has been tested in different event-driven environments and shows the effectiveness of boosting the performance of event handling.

In order to analyze the performance of an event-driven application, a mathematical model for parallel event-driven programming is developed. This model reveals how factors such as asynchronization scale and parallelization scale influence event-driven performance. This theoretical guidance can help programmers to tune the performance for the event-driven system and make proper decisions on choosing the number of asynchronous workers and the number of parallel threads for each event handler.

Extensive experimental tests and benchmarks have been conducted and the results show that Pyjama is both a useful and productive tool for parallelizing Java code. The benchmarks show the Pyjama directives are effective in parallelizing work, and that the speedups are comparable to C/C++ OpenMP implementations. Different aspects of the evaluation demonstrate that using the new proposed virtual target programming model is efficient to boost the performance of event-driven applications.
ACKNOWLEDGMENT

It would not have been possible to complete this thesis without the help and support of a number of people.

First and foremost, I would like to sincerely express my gratitude to my supervisors Dr. Nasser Giacaman and Dr. Oliver Sinnen for their guidance and inspiration over the past few years. Without their constant supervision, encouragement and great support throughout the years, I would not have been where I am now. Nasser, I learned a lot from you. It would be a great honor being your first PhD student! Oliver, thank you for helping me to remove any doubt along with my PhD study. Your attitude and passion to the research inspired me and guided me to be a qualified researcher.

I thank my uncle Professor Aiguo Patrick Hu and his family, for caring for me throughout the days I lived in Auckland. Thank you - Aunt Fiona Fang Zhao! You always invited us for dinners and gave all I needed with regard to life. I also thank my cousins and friends, Alice Hu, Belinda Hu, Peiran Quan and Lijun Wang.

In addition, I am grateful to all members of the Parallel and Reconfigurable Computing group (PARC) for their generosity in sharing knowledge and experience in work and life. I thank my colleagues in PARC including Mr. Mostafa Mehrabi and Mr. Haomiao Wang. I would like to express my many thanks to Mr. Fung Yang and Mr. Emmanuel Baguia for their laboratory assistance.

I am also very grateful to my friends and colleagues in the Department of Electrical and Computer Engineering (ECE) including Dr. Liang Huang, Dr. Zhenmin Li, Dr. Howard Wang, Mr. Lei Zhao, Mr. Jianjie Tian, Ms. Yuan Liu, Mr. Yuan Song, Mr. Liming Qiu, Mr. Di Wang, Ms. Su Zhang, Mr. Zhijia Wang and other peers for their companionship and various help.

Last but not least, I would thank my beloved family. Thank you, my dearest parents! I am the luckiest boy, because of you, who gave me life and supported my life. Thank
you my brother Jing Fan. Your unconditional emotional support and care are always
the strongest backing for me to get through all the sorrows and difficulties.
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

**Exception Handling with OpenMP in Object-Oriented Languages.**

Published in the Proceedings of 11th International Workshop on OpenMP (IWOMP 2015).

DOI: 10.1007/978-3-319-24595-9_9

The materials of the work are used in Chapter 3 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>The concepts, development of the models, implementation, evaluation and manuscript writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;80%</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostafa Mehrabi</td>
<td>Proofreading and check consistency of the work.</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostafa Mehrabi</td>
<td></td>
<td>6/12/2017</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>13/12/2017</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td></td>
<td>13/12/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Supporting Enhanced Exception Handling with OpenMP in Object-Oriented Languages.

Published in International Journal of Parallel Programming (2016).

DOI: 10.1007/s10766-016-0474-x

The materials of the work are used in Chapter 3 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>The concepts, development of the models, implementation, evaluation and manuscript writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;80%</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostafa Mehrabi</td>
<td>Proofreading and check consistency of the work.</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostafa Mehrabi</td>
<td></td>
<td>6/12/2017</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>8/12/2017</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td></td>
<td>13/12/17</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Asynchronous OpenMP Tasking with Easy Thread Context Switching and Pool Control
Published in the Proceedings of 13th International Workshop on OpenMP (IWOMP 2017).
DOI:10.1007/978-3-319-65578-9_15

The materials of the work are used in Chapter 5 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Suggestions on the concept, proofreading and check consistency of the work.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;80%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>13/12/17</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td></td>
<td>13/12/17</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Towards an Event-Driven Programming Model for OpenMP.

Published in the Proceedings of 45th International Conference on Parallel Processing Workshops (ICPPW 2016).

DOI: 10.1109/ICPPW.2016.44

The materials of the work are used in Chapter 4 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>The concepts, development of the models, implementation, evaluation and manuscript writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;80%</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>8/12/17</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td></td>
<td>13/12/17</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Evaluating OpenMP Implementations for Java Using PolyBench.

Published in the Proceedings of 12th International Workshop on OpenMP (IWOMP 2016).

DOI: 10.1007/978-3-319-45550-1_22

The materials of the work are used in Chapter 6 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>The implementation of the compiler, experiments and benchmark runnings and manuscript writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;70%</td>
</tr>
</tbody>
</table>

## CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rui Feng</td>
<td>Implementation of the benchmarks.</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

## Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rui Feng</td>
<td></td>
<td>6 December 2017</td>
</tr>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>13/12/17</td>
</tr>
<tr>
<td>Nasser Giacaman</td>
<td></td>
<td>13/12/17</td>
</tr>
</tbody>
</table>

_Last updated: 19 October 2015_
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

**Supporting Asynchronization in OpenMP for Event-Driven Programming.**

Published in the journal Parallel Computing (ParCo).

The materials of the work are used in Chapter 4 of this thesis.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>The concepts, development of the models, implementation, evaluation and manuscript writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>&gt;80%</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
<tr>
<td>Nasser Glacaman</td>
<td>Supervision, advise, guidance and manuscript revision.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Sinnen</td>
<td></td>
<td>13/12/17</td>
</tr>
<tr>
<td>Nasser Glacaman</td>
<td></td>
<td>13/12/17</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
CONTENTS

1 INTRODUCTION 1
  1.1 Shared-memory Parallel Programming 1
  1.2 OpenMP 2
  1.3 Thesis Motivation 2
  1.4 Thesis Contribution 5
    1.4.1 A Java OpenMP compiler with more high-level features 5
    1.4.2 A proposal and implementation of OpenMP exception handling for Java 6
    1.4.3 A programming model for event-driven and its performance quantification 7
  1.5 Thesis Structure 8

2 BACKGROUND 9
  2.1 Shared-Memory Parallel Systems 9
    2.1.1 Hardware Architecture 9
    2.1.2 Memory model 11
  2.2 Shared-Memory Parallel Programming 11
  2.3 OpenMP 13
    2.3.1 Directive-based parallel programming model 14
    2.3.2 OpenMP for Java 15
    2.3.3 Gaps between OpenMP and Object-oriented Programming Concepts 15
  2.4 Pyjama 17
    2.4.1 An overview of the Pyjama framework 17
    2.4.2 Pyjama extended features 18
    2.4.3 Pyjama Tasking 21
4.4.1 Compilation 84
4.4.2 Runtime 88
4.5 Evaluation 91
  4.5.1 Java GUI event handling 91
  4.5.2 Responsiveness of the EDT 93
  4.5.3 Web service event handler 98
  4.5.4 Android application 99
4.6 Related Work 100
  4.6.1 Asynchronization 100
  4.6.2 Task-based Parallelism 102
4.7 Summary 103

5 Performance Modeling of Event-Driven Programs with OpenMP 105
  5.1 Introduction 106
  5.2 Performance measurement of event handling execution 107
    5.2.1 Flow time of event handling 108
    5.2.2 Processing events in a multi-threaded environment 108
    5.2.3 Modeling of the parallel event-driven system 109
    5.2.4 Discussion of the model 112
  5.3 Evaluation 115
  5.4 Summary 120

6 Java OpenMP Benchmarks and Evaluations 123
  6.1 Introduction 124
  6.2 PolyBench for Java OpenMP 125
  6.3 Evaluation 125
  6.4 Results 129
    6.4.1 Long runtimes 129
    6.4.2 Medium runtimes 131
    6.4.3 Short runtimes 133
    6.4.4 Other Observations 133
  6.5 Related Work 134
  6.6 Summary 135

7 Conclusions 137
  7.1 Future works 138
    7.1.1 Worksharing tuning 138
7.1.2 Incorporating asynchronous I/O to virtual target 139
7.1.3 Software transactional memory 140
7.1.4 Event-driven online optimization 140

A Pyjama 2.0 Syntax Quick Reference 143

Bibliography 155
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Illustration of the fork–join model of parallel programming (Source: Wikimedia Commons [1]).</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2</td>
<td>An overview of the Pyjama framework.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The task directive in Pyjama.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Extended cancellation directive.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Different uses of cancel directive. (a) cancel parallel local Only single thread quits the innermost parallel region; (b) cancel parallel global Entire thread group quits current parallel region; (c) cancel for local Single thread quits current work-sharing for-loop, but continues when other threads finish this for-loop iteration; (d) cancel for global All threads quit the current worksharing for-loop and continue with the following statement.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 6</td>
<td>A legal local exception handling try block cannot traverse any line as indicated.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Flowchart of exception handling within an OpenMP parallel region.</td>
<td>38</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Pure overhead comparison in two systems.</td>
<td>50</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Absolute time of synchronization overhead in a Java version of OpenMP (Pyjama).</td>
<td>52</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Absolute execution time of EPCC.</td>
<td>52</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Overhead of TC/TC&amp;CCP evaluations onto different OpenMP directives.</td>
<td>53</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Comparison of global exception handling overhead in two systems.</td>
<td>54</td>
</tr>
</tbody>
</table>
Figure 13 Absolute quitting time when an exception happens in parallel execution. 54

Figure 14 In an event-driven application, the EDT plays the role of the main thread responding to events. An essential requirement is to maximize the idleness of the EDT, so programmers are required to transform single-threaded event processing to multi-threaded event processing to increase the responsiveness of the EDT. 63

Figure 15 An example of event handling logic, where a time-consuming computation involves background components (S1 and S3), with a foreground progress update (S2), before a concluding foreground computation (S4). 64

Figure 16 Java asynchronous programming with SwingWorker. 65

Figure 17 C# AMP-style programming. 66

Figure 18 Extended target directive. 68

Figure 19 Conceptual difference between virtual target and device target. 70

Figure 20 Semantic example of using virtual target directive. 71

Figure 21 Different asynchronous modes, by using different asynchronous-property-clauses. 74

Figure 22 Definition of async-call directive, for the purpose of calling an async function in different ways. 76

Figure 23 Semantic example of using await directive. 77

Figure 24 Definition of async directive, for the purpose of annotating an async function. 78

Figure 25 The control flow of using awaits directive. 78

Figure 26 Demonstration of OpenMP tasking (a) and virtual target (b). 83

Figure 27 An overview of the TargetTask Class. 85

Figure 28 Average event response time, as a proportion of the sequential version, using different offloading approaches and computational kernels (lower is better). 92

Figure 29 5-second response count under different request work loads (higher is better). 94

Figure 30 The EDT execution profiling on the searching scenario. 96
Figure 31   Cumulative Latency Distributions. 98
Figure 32   Throughput scaling comparison between Jetty and Pyjama. 98
Figure 33   The work flow of the experimental app. 99
Figure 34   A parallel event dispatching model that has multiple asynchronous processors and parallel processors. 107
Figure 35   The theoretical relationship between processor utilization $\rho$ and the mean event-handling flow time $W$ in an event-driven system. 113
Figure 36   The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 0.8, \mu = 1$). 114
Figure 37   The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 10, \mu = 12$). 114
Figure 38   Minimum speedups $\epsilon(N_p)$ should be achieved that make the same $W$ when the scale is applied to asynchronization. 115
Figure 39   The mean flow times (ms) of four kernels implemented by three different asynchronization scale distributions. 117
Figure 40   64-processor distribution and its mean flow time (ms), categorized by different kernel handlers. 118
Figure 41   Parallel speedup of four kernels and the minimum speedup threshold for $\rho = 0.8$. 119
Figure 42   Performance comparison of two types of distribution between asynchronization and parallelization with total amount of 64 processors. 119
Figure 43   Performance comparison of two types of distribution between asynchronization and parallelization with total amount of 24 processors. 120
Figure 44   Selected benchmark cases (PL2mm, PL3mm and PLsymm) where Java implementations have good performance and scalability (comparing to C). 130
Figure 45   Selected benchmark cases (PLtrmm, PLsyrk and PSfddt_2d) where Java implementations have comparable performance with C versions, but scalability is not very good. 131
Figure 46   Selected benchmark cases (PLtrisolv, PLatax, and PLmv) where Java implementations have bad performance and scalability. 132
Figure 47 Benchmark cases (PLdynprog and PMreg_detect) show their unscalability of parallelisation. 134
<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 1</td>
<td>Pyjama GUI aware directives.</td>
<td>19</td>
</tr>
<tr>
<td>Listing 2</td>
<td>Parallelizing the for-each loop by using Pyjama.</td>
<td>20</td>
</tr>
<tr>
<td>Listing 3</td>
<td>Parallelize the iterator-based loop by using Pyjama.</td>
<td>20</td>
</tr>
<tr>
<td>Listing 4</td>
<td>Custom reduction using Redlib.</td>
<td>20</td>
</tr>
<tr>
<td>Listing 5</td>
<td>Try-catch mechanism that does not syntactically and semantically conform with the OpenMP specification.</td>
<td>30</td>
</tr>
<tr>
<td>Listing 6</td>
<td>Syntactically conforms with OpenMP specification, but semantically it has a defect.</td>
<td>30</td>
</tr>
<tr>
<td>Listing 7</td>
<td>An example of using the throw clause.</td>
<td>39</td>
</tr>
<tr>
<td>Listing 8</td>
<td>A demo code using the neglect_exception clause to simplify the recovery procedure when an uncaught exception happens inside a parallel region.</td>
<td>40</td>
</tr>
<tr>
<td>Listing 9</td>
<td>Conventional approach of handling global exception.</td>
<td>42</td>
</tr>
<tr>
<td>Listing 10</td>
<td>Pyjama’s new approach of handling a global exception.</td>
<td>43</td>
</tr>
<tr>
<td>Listing 11</td>
<td>An example of using OpenMP tasks.</td>
<td>82</td>
</tr>
<tr>
<td>Listing 12</td>
<td>An example of using virtual targets.</td>
<td>83</td>
</tr>
<tr>
<td>Listing 13</td>
<td>The search listener function implementation.</td>
<td>95</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Two types of synchronization point in OpenMP.</td>
<td>35</td>
</tr>
<tr>
<td>Table 2</td>
<td>OpenMP runtime exceptions.</td>
<td>48</td>
</tr>
<tr>
<td>Table 3</td>
<td>Key values of overhead measurement in two systems.</td>
<td>51</td>
</tr>
<tr>
<td>Table 4</td>
<td>Runtime functions to create virtual targets in Pyjama.</td>
<td>80</td>
</tr>
<tr>
<td>Table 5</td>
<td>Comparison between OpenMP tasking and target virtual.</td>
<td>82</td>
</tr>
<tr>
<td>Table 6</td>
<td>Interval types and their represent colors.</td>
<td>97</td>
</tr>
<tr>
<td>Table 7</td>
<td>A comparison of three approaches, with regard to programming effort and performance.</td>
<td>101</td>
</tr>
<tr>
<td>Table 8</td>
<td>Computational kernels and their arrival rates and sequential service rates.</td>
<td>116</td>
</tr>
<tr>
<td>Table 9</td>
<td>List of implemented PolyBench benchmarks, with their sequential execution times (ms).</td>
<td>126</td>
</tr>
<tr>
<td>Table 10</td>
<td>List of implemented PolyBench benchmarks, with their sequential execution times (ms) (continued).</td>
<td>127</td>
</tr>
<tr>
<td>Table 11</td>
<td>Four types of compilation of OpenMP code.</td>
<td>128</td>
</tr>
</tbody>
</table>
LIST OF ALGORITHMS

Figure 3.1  Adaptable synchronization barrier. 45
Figure 3.2  The cancellation signal processing in runtime. 47
Figure 4.1  Target block code execution. 89
Figure 4.2  Post target block to different types of virtual target. 90
INTRODUCTION

SHARE-MEMORY PARALLEL PROGRAMMING

Traditionally, high-performance computing was only developed and used for large-scale data processing problems, due to these solutions relying on specialized knowledge, computing environments and expensive systems and hardware. Because of the computational limitations of a single computer, distributed systems are extensively researched. A distributed system consists a group of computers in which the computers communicate and coordinate via network by passing messages [2]. Some new message passing mechanisms are proposed and implemented such as Remote Direct Memory Access (RDMA) [3], to improve the performance of communication among subsystems. Even though so many parallel programming systems and the language interfaces based on them have been developed, most of them are not widely used. One of the reasons behind this is that most efforts are proposed as new languages and approaches, rather than supporting existing approaches [4]. Another reason is the intrinsic complexity of developing parallel programs in distributed systems.

Nowadays, multi-core computing devices have become prevalent, ranging from high-performance blade servers to ordinary PCs, and to mobile devices. These devices usually have many cores and the CPUs share a global addressed space. Compared with distributed systems, a shared-memory system is more efficient on passing data between computational units, and also it is relatively easier for the programmers to write parallel programs. Recent research contends that a single shared-memory machine is sufficient for solving many problems in large-scale computing [5].

Since shared-memory multi-core devices are everywhere. There is a trend that programs and applications are written in concurrent ways to benefit from parallel execution.
However, parallel programming is much more difficult, compared with writing sequential programs. Even experienced programmers may make counter-intuitive mistakes in coding, leading to a system malfunction or an incorrect program execution. That is the main reason why so many parallel programming frameworks and languages are explored and invented, to facilitate shared-memory parallel programming.

**OPENMP**

Among all of the existing shared-memory programming models, OpenMP has gradually gained popularity in shared-memory parallel programming due to its simplicity. OpenMP operates using a fork-join threading model (Figure 1), and parallelizes code by adding compiler directives. One of the salient advantages of the OpenMP interface is that it allows programmers to convert the control flow into a parallel version without breaking the original order and structure of the sequential code. Most of the low-level code is generated by compilers underneath, including explicit thread creation and thread-local memory allocation. This approach helps programmers focus more on programming logic and avoid minor low-level mistakes.

**THESIS MOTIVATION**

Although OpenMP has many advantages, it is still rare to see this model being used for a wide range of applications. This phenomenon is due to the fact that current OpenMP
specification stresses numerical low-level parallelism, using traditional procedural lan-
guages such as C and Fortran.

Even though the evolution of OpenMP has made the OpenMP standard progressively
comprehensive for shared-memory applications, the framework still has some way to
go before it is widely used for general software development. For example, in modern
software development, high-level object-oriented programs widely use abstract data
structures, known as objects. The current OpenMP standard mainly addresses primitive
type data handling, and has little specification for class type data. Also, as for loop
parallelization, the standard OpenMP specification cannot cater for high-level loops
such as for-each loops and iterator-based loops. Furthermore, even though OpenMP
already provides customized reduction, it is counter-intuitive and hard to bind with
object-oriented functions. The defects are not limited to those aspects, but it is enough
to show the imperfection of OpenMP for high-level programming.

Since object-oriented languages such as Java and C++ have their intrinsic advantages
for large scale, cooperation-based programming, they have become dominant in indus-
trial software development. In light of this situation, it is highly desirable to establish a
comprehensive parallel programming solution for object-oriented languages, providing
novel programming paradigms and optimization tools to give a better parallel program-
ing experience for programmers. This thesis argues that a good programming model
should:

- Be an easy adoption for software engineering.
- Release programming burdens, and reduce development time.
- Increase performance and computing efficiency.

This research tries to identify issues with the existing OpenMP specification and adapt
it to suit object-oriented languages from a software engineering point of view. The
original OpenMP interface will also be extended and improved for high-level software
development.

There is a need to develop an advanced interface to cover the issues using OpenMP-
like directives in object-oriented languages. This research, more specifically, is an exten-
sion of the OpenMP standard for Java language, trying to give a solution to make Java
benefit from OpenMP. All the potential ideas will be applied to Pyjama, which is an
OpenMP-like parallelization tool for Java.
Two important aspects under the scenarios of object-oriented parallel programming are especially addressed: control flow and programming model.

**Control flow.** In sequential programming, the control flow is trivial because it will always be a single thread executing the program. However, when it comes to parallel programming, programmers have to care about multiple control flows since different threads have to cooperate and synchronize with each other to achieve the correct execution. The same problem exists when using OpenMP. Two aspects raise their importance and become the main obstacles in parallel programming with OpenMP. The first is error recovery mechanism and the second is flexible thread cancellation.

In Java, exception handling is widely used, which is an important part for software robustness. When exception handling is used with OpenMP parallel code, things get ambiguous because there is no specification on how exceptions work with parallel regions. Since it is necessary to support exception throwing and catching for robust parallel programming, this objective is mainly for the combination of Java exception handling with OpenMP directives. It involves the formalization of the semantic meaning when throwing or catching exceptions inside parallel regions, or spanning parallel regions, and building the runtime support for exception handling in a multi-threaded environment.

A flexible thread cancellation in a multi-threaded environment is of equal importance. In some programming scenarios, a thread may require termination if it encounters an error or a specific situation. As in parallel environment, a termination could be a global termination that stops the entire parallel processing, or a local termination that only the encountering thread stops. How to arrange all types of thread cancellation requires OpenMP to have a flexible cancel directive and its runtime support.

**Programming model.** OpenMP is designed for batch-like scientific computing, where the execution is defined by the given input. During the execution it does not require further external inputs. For this type of programming, the fork-join model is the most suitable parallelism model. However, another computing model widely used in modern applications that OpenMP does not cover is the event-driven model. Event-driven is the dominant model for Graphical User Interface (GUI) applications, and also for many serving applications. Because of the interactive nature of this model, computational speedup is not the only factor to influence the efficiency of the entire system.

This model also needs drawing the benefit from parallelism. Under this scenario, concurrency and asynchronization, instead of parallelization, are the key factors to
achieving good performance and user experience. This challenge lies under the premise that the new programming model does not violate the philosophy of OpenMP, and how this model can be adaptable for a wide range of event-driven applications in harmony with the fork-join model.

Another aspect is the evaluation of the parallel event-driven application. Traditional parallel benchmarks have limited usefulness with regards to event-based applications, because the measurement criteria are different. Instead of measuring the speedup for traditional parallel problems, developing a performance model for parallel event-driven system is also of high necessity.

**THESIS CONTRIBUTION**

This research developed a new tool to improve programmers productivity and efficiency on object-oriented parallel programming. New concepts and ideas are invented and implemented in Pyjama, a Java implementation of OpenMP-like directives and runtime routines.

*A Java OpenMP compiler with more high-level features*

This research presents Pyjama, a Java OpenMP compiler and its runtime. By adhering to the standard OpenMP specification, the sequential code can be quickly parallelized by adopting OpenMP-like comments. In addition, some new features are invented to facilitate high-level object-oriented programming patterns. The main extended features are described below:

**Loop parallelism:** Traditional OpenMP only supports very basic integer indexed for-loops. Advanced loops are prevalently used in high-level languages, and existing loop support is far from sufficient when programmers wish to employ more advanced for-loops such as iterators or for-each loops over data collections. This improvement makes current OpenMP worksharing directives support more loops, from original integer indexed for-loops to iterator-based loops and for-each loops. Also, new cancel directives are extended, to make it possible for programmers to stop iterations midway in the execution of a parallel loop.

**GUI-support:** The traditional OpenMP specification does not take Graphic User Interface (GUI) into consideration, therefore there is a lack of parallelism employed in
desktop GUI applications and mobile platforms such as Android. In Pyjama, parallelization is expanded to support the Event Dispatching Thread (EDT), and introduces new concepts to facilitate GUI parallel programming. This aspect involves auxiliary directives and runtime support to facilitate quick event-based parallel application development. Some preliminary work has already been undertaken to integrate support for GUI extensions in an OpenMP-like manner. The directives have been added to support parallel execution when invoking the parallel region on the EDT. The idea is to execute a block of code by a background thread or a group of threads, thus allowing the EDT to go directly back to the event loop in order to handle other events. This approach enhances the responsiveness of the user interface.

**Custom reductions:** Even though the standard OpenMP specification already provides customized reductions, its usability is non-intuitive, because the syntax of declaring a customized reduction requires special identifiers to take part in, which makes the declaration difficult to understand. In Pyjama, the reduction interface is consistent but without losing its usability. When Pyjama presents a reduction operation, the operator can be a primitive such as plus or minus, or it can be a complex function call. Programmers can define functions as reduction operations, as long as the function conforms with the pre-defined format. Aside from that, an extensive library of reduction operations, including nested reductions and reductions for aggregate data types is provided for programmers.

*A proposal and implementation of OpenMP exception handling for Java*

The current OpenMP standard lacks support for essential programming features such as mechanisms for error recovery. Exception handling is such a mechanism in object-oriented programming. In languages such as Java, the concept of exception handling has been an integral aspect of the language since the first release. For OpenMP to be truly embraced within this object-oriented community, exception handling needs to be given some attention. The official OpenMP standard has little specification on error recovery, as the challenges of supporting exception-based error recovery in OpenMP extends to both the semantic specifications and related runtime support.

This thesis proposes a systematic mechanism for exception handling with the co-use of OpenMP directives based on Java. The concept of exception handling with OpenMP directives has been formalized and categorized. Hand in hand with this exception
handling proposal, a flexible approach to thread cancellation is also proposed (as an extension to OpenMP directives) that supports this exception handling within parallel execution. The improvement of control flow gives programmers the flexibility and ease to control the execution.

The runtime support and its implementation are discussed. The evaluation shows that while there is no prominent overhead introduced, the new approach provides a more elegant coding style which increases the parallel development efficiency and software robustness.

**A programming model for event-driven and its performance quantification**

This thesis proposes an asynchronous programming model based on the philosophy of OpenMP, which does not require code restructuring of the original sequential code. The GUI concurrency concept is be extended to a universal event-driven scenario where the Event Dispatching Thread (EDT) monitors an infinite loop and constantly processes a series of events or computation requests. This concept has been implemented into Pyjama, to help programmers write OpenMP code efficiently when developing GUI applications or any other event-based applications. This asynchronous programming model is complementary to the existing OpenMP fork-join model. The coexistence of the two models makes OpenMP able to handle a wider range of parallelism problems, and the evaluation shows the effectiveness of boosting the performance.

**Publications**

The following publications incorporate material from this thesis:


THESIS STRUCTURE

The structure of this thesis is as follows: Chapter 2 reviews the background, and motivates the extension of OpenMP for object-oriented programmings. Chapter 3 describes enhanced exception handling for OpenMP in object-oriented languages. Chapter 4 introduces the concept of supporting asynchronization for facilitating event-driven programming. Chapter 5 discusses the performance model of a parallel event-driven system. Chapter 6 evaluates the performance of Pyjama by demonstrating the results of some benchmarks. Chapter 7 summarizes the conclusions obtained from this thesis. Future research directions are also provided.
BACKGROUND

This chapter systematically summarizes the background of shared-memory parallel programming and OpenMP. The main purpose of this chapter is to reveal every noticeable aspect when doing shared-memory parallel programming and writing OpenMP code. Four aspects in relation to the evolution of shared-memory object-oriented programming are listed, spanning from theoretical modeling research to practical engineering techniques.

The content of this chapter finally narrows down to a brief introduction to Pyjama. In this section, the Pyjama compiler framework is demonstrated. Some unique features which distinguish Pyjama from traditional C/C++ OpenMP are also presented.

SHARED-MEMORY PARALLEL SYSTEMS

Hardware Architecture

For a conventional sequential computing model, von Neumann architecture has won its dominant position over the past several decades. The main advantage of the von Neumann machine is the abstraction of control flow, which makes it possible for computers to store both data and instruction flow in memory space randomly. Abstraction of control enables programmers to represent programs in software-level and makes it independent from the hardware implementation.

The von Neumann model works fine in sequential machines and the efficiency has improved over the last 30 years with the boost of processor clock rates. However, when this model applies on a multi-processor machine, many problems emerge. The biggest problem is the synchronization of memory access and control flow between
different processors. Modern compilers’ optimization techniques cannot be applied for multi-core programs because most compilers may change the instructions without programmers’ awareness. Modifications include changing the sequence of code, removing redundant operations, loop optimization and so on. All those changes are based on the principle that machine code works as the programmer expects, but the procedure itself cannot be ensured to be the same as the code indicates.

This situation makes it difficult for programmers to write parallel source code. Regarding to sequential coding, people do not need to know how the compiler interprets their codes and how they are executed on the hardware. In contrast, when people are writing source code for multi-processors, they have to be aware of the potential unexpected hardware behavior which may break up the communication and synchronization between processors.

There is no universal model for shared-memory multi-core machines. The most well-know and simplest is the PRAM model [6]. However, the PRAM model is no more than an ideal model and cannot be used for practical situations, because PRAM does not take cache systems into account which is an indispensable part for modern computer architecture. Other models have emerged as well such as LogP [7] and BSP [8], but all of those models have their shortcomings with respect to different scenarios. The diversity of parallel machine models makes it harder for people to write parallel code efficiently because they have to notice that the differences between hardware systems will influence the implementation of code, especially when writing with low-level programming languages such as C.

Practically, for multi-core architectures, multiple processors do not directly get access to the cache. Instead, each processor has its own cache. Because of this locality, each processor is able to process instructions and data at very high speed as long as they have no data being shared with each other. But once processors communicate via shared data, the coherence between caches becomes important for the correctness of the program. The most commonly used protocol is MESI [9], in which cache states are divided into four different statuses: modified, exclusive, shared, and invalid. The MESI protocol ensures different processors are able to get coherent memory context.

For hardware, another important aspect is the universal synchronization primitives (hardware instruction) provided to support complex synchronization data structure, from locks to semaphores to monitors. There are basically two types of hardware synchronization instructions: One is CMPXCHG, also known as CAS (compare-and-swap),
supported by Intel, AMD and Sun architectures; the other is LL/SC (Link-load/Store-conditional), supported by PowerPC, MIPS and ARM architectures.

Memory model

Memory model describes the constriction when a group of threads interact onto shared data via memory. The strictest model is Sequential Consistency, which was enacted by Lamport [10, 11]. Sequential Consistency requires two limits on code execution: a) all instructions are executed in order; b) each write operation is instantaneously visible for the entire system. For programmers, the advantage is, this model is easy to reason about the behavior of code, making it intuitive for programmers to compose the expected parallel code. However, the drawback of Sequential Consistency is that it bans all possibilities of code optimization for compilers. Lack of efficiency makes this memory model unrealistic to put into practical use.

In order to strike a balance between performance and intuitive programming, several relaxed consistency models are developed [12, 13, 14]. As for Java, since the bytecode is executed on a virtual machine, there is a special specification for the Java memory model that defines the legal behavior for multi-threaded execution [15, 16, 17, 18].

The non-deterministic nature of multi-core programs could be the most prominent barrier for programmers to write correct code. Because the parallel code may execute at any legal interleaving in accordance with the memory model and real-time scheduling policy, the program runtime could behave differently from time to time. The possibility of interleaving increases exponentially with the increased portion of parallel code. That makes it difficult to debug parallel programs since it is impossible for programmers to test all possible interleaving results. Many studies have been undertaken to reduce the non-determinism of parallel execution, such as deterministic scheduling [19, 20, 21, 22] and deterministic programming style [23].

SHARED-MEMORY PARALLEL PROGRAMMING

Semantic research mainly focuses on the issue about the expression of parallel computing. The goal of parallel semantic research is to give a well-established semantic meaning for programmers to express parallel instructions. There are many semantic research investigations for parallel programming.
Theoretically, there are mainly two types of semantic models for parallel programming: Memory Sharing and Message Passing. In the shared memory model, data is directly shared by sharing its address. The advantage of directly sharing memory is efficiency and simplicity. Different threads write or read shared variables asynchronously. However, the drawback is proper protection, requiring a synchronization mechanism for correct execution. The other model is based on message passing, where processes or threads communicate with each other by passing messages. There are no data-races since different threads work within their own exclusive memory space.

Java supports a concurrency model for users by explicitly creating threads. In light of the object-oriented principle, threads are technically objects. One thread is created by implementing the Runnable interface or extending the Thread class. Thread behavior is defined by overriding the run() method. In addition, Java also provides the java.util.concurrent package, which contains a variety of types of useful concurrent interfaces and classes for programmers to develop parallel code. Support includes several high-level interfaces such as the ThreadPoolExecutor and fork-join framework, and different types of synchronization, locks, mutex primitives, as well as a group of thread-safe data structures. The collection of tools makes it easier for programmers to write parallel code, but programmers still need a lot of expertise to utilize these libraries.

A great amount of new concurrent programming techniques and tools are available, such as Intel Threading Building Blocks [24], Cilk [25, 26], X10 [27], Chapel [28], Habanero [29, 30], Parallel Task [31], and OpenMP [32]. Microsoft provides different parallel programming tools both for C++ and C#. As for C++, Parallel Patterns Library (PPL) can be used, which is similar to the OpenMP interface. Task Parallel Library (TPL) is for C# and the .NET Framework. New languages such as Google’s Go language have emerged, supporting concurrent programming built into the language. In Go, starting one routine is easy by adding the modifier go before one function or procedure. Also, Go uses channels between routines to pass data. The benefit of using channels is in avoiding data races. Cilk [26] is a research language supporting parallel semantics. The biggest principle behind the design of the Cilk language is that the programmer only should be responsible for exposing the parallelism, identifying elements that can be executed in parallel. However, all the low-level work, including scheduling, mutual-exclusion, and load-balancing are designated to the source-to-source compiler and runtime routines.

Functional programming languages have their intrinsic advantage for high-concurrency programming, because of the feature of immutable data. Erlang dominates parallel pro-
cessing in some industrial fields [33]. High concurrency is achieved by creating many light-weighted processes. Data sharing is achieved by message passing.

Pthread is POSIX standard, providing a group of interfaces for C/C++ programming language. Pthread entails programmers make very fine-grained synchronization and data sharing. However, the primitive nature of Pthread allows data to be shared among threads, but this may cause problems such as data races, deadlock or livelock, which make parallel programming error-prone and very difficult to debug [34]. As a counterpart, another parallel programming model, known as Message Passing, eliminates data races by sending or receiving messages to or from other processes. Different processes communicate via a series of messages, instead of directly operating on the same memory space. The Actor model [35] and Communicating Sequential Processes (CSP) [36] are two typical implementations of this idea. Though there is already a lot of research done in developing frameworks or extensions of this model for specified language (e.g. [37], [38] and [39]), the semantic change is inevitable and the code modification is still a nontrivial work for programmers to put into painless use.

OPENMP

OpenMP [32] is a cross-platform standard for shared-memory programming, using a simple fork-join model. By adding extra directives, sequential code is easily converted into a parallel version. The advantage of OpenMP is that it requires only incremental changes to sequential code, freeing programmers from demanding refactoring work for parallelism. Also, with proper extension and runtime support, it is possible for this model to support distributed computing [40].

Traditional parallel code design, in which thread callbacks are implemented separately, has several drawbacks. The most salient defect of standalone thread composing is the lack of parallel context. Since threads are implemented for their own purposes, data sharing and inter-thread synchronization is implicit. As a consequence, it is a nontrivial effort for both programmers and compilers; it demands they are aware of important aspects for the correctness of the parallel program, including data sharing, mutex, signal and wait, and so on. More specifically, for developers, separation of thread context forces people to notice the potential influence to other threads in terms of rewriting values on shared data, for the reason that for shared variables and data structures, mutex operations are necessary to achieve correct computing results. For compilers,
Sequential code optimization techniques have limitations because they could break the inter-dependency among threads that programmers may expect. This leads to a dilemma between correctness and efficiency.

OpenMP has its intrinsic merit, through its directive-based parallelization. Its representation of parallelism is based on original sequential code, without any changes to the initial purpose of the code. By adding directives, OpenMP weakens the concept of thread individualism, and enhances the awareness of synchronization and synergy for certain processes as sequential code describes in a parallel region. As a result, parallelizing code using OpenMP holds its advantage of less code reconstruction. At the same time, the code itself represents the parallelization property. From the development point of view, explicit parallelism annotations remind programmers to take care of parallelization security checks, including proper task dividing and thread-safety checks. From the optimization point of view, explicit parallel code representation enables compilers make exclusive optimization for parallel regions. This advantage is especially prominent in parallelization of iterations or loops.

**Directive-based parallel programming model**

Apart from OpenMP, there are many other parallel programming models available for programmers to choose from, according to different hardware platforms and programming purposes. For example, the OpenACC [41] Application Program Interface, which has a similar programming style to OpenMP, provides a collection of compiler directives to specify loops and regions of code to be offloaded from a host CPU to an attached accelerator (e.g. GPU, APU). OpenUH [42] is an OpenMP implementation for clusters of SMPs. It uses the OpenMP interface, but the back end is re-implemented with further portability and optimizations. OmpSs [43] draws the advantages of OpenMP and StartSs [44], and incorporates the OpenCL and CUDA kernels, aiming at heterogeneous multi-core architectures. Patty [45] is a pattern-based parallelism technique. By addressing software engineering, Patty provides several parallel patterns and programmers can easily parallelize code by adding annotations.
OpenMP for Java

Although there is no official OpenMP standard for the Java programming language, several research efforts have been undertaken to support OpenMP-like implementation in Java. Most of them only implement part of the OpenMP standard. JOMP [46] is a source-to-source compiler, converting Java code with OpenMP directives to standard Java code by using built-in Java thread support. JaMP [47] is designed to support distributed shared memory platforms, translating sequential OpenMP-added code to parallel code, which uses the Jackal framework. Cook’s approach [48] is directly implementing a Java library, simulating the usage of OpenMP. The advantage of simulating OpenMP as a Java library is its simplicity, because there is no necessity to develop a compiler to convert the code. However, the drawback is that it needs code refactoring and translation from the original sequential code. AOmpLib [49] is an aspect-oriented library that imitates OpenMP for Java using AspectJ.

Gaps between OpenMP and Object-oriented Programming Concepts

This research investigated 100 open source C/C++ projects in which OpenMP directives were used (as were retrieved by OpenMP Hub Ohloh\(^1\) source code search engine). Among those projects, 69 used OpenMP directives only as part of their test suit. Because those projects are created for special compiler developments or for ad-hoc platforms or infrastructure, the test suit is only an auxiliary part for correctness concern for the software, so they were omitted for the purposes of this study. In contrast, the remaining 31 projects used OpenMP directives for actual functionality of the software. Those projects include scientific applications (e.g., SAGA GIS, tmlQCD), image processors (e.g. Blender, ImageMagick), and various kinds of computational libraries (e.g., Madagascar, Kranc, MPACK). With regards to the OpenMP directives used, most used `omp parallel for`, which accounted for approximately 80% of the total number of directives. In most cases, this directive was used for conventional for-loops to boost matrix or vector computations, which are typical intensive numeric computations. It was rarely found that the directive was used for more advanced data structures or objects. It was also noticed that OpenMP is used in limited types of applications which indicate that the current OpenMP interface is not very suitable for general purpose software devel-

\(^1\) https://www.openhub.net/
opment, particularly for highly interactive and desktop applications that involve event handling.

The first major difference between scientific computing and general user-oriented applications lies in the programming languages. Scientific applications are typically written in speed efficient languages that tend to be low level, for example C and Fortran. Desktop applications, on the other hand, tend to be developed using high level and object-oriented languages to promote a software engineering approach to programming. For example, Qt (C++) is used for the K Desktop Environment, Google Earth and Skype. Windows applications are typically developed using the .NET Framework, Mac OS X applications are typically developed using Objective-C, while Java is inter-operable on all operating systems (in particular Android developments).

The second major difference lies in the control flow of these applications. Scientific applications tend to be batch-type and compute-intensive. Given input parameters, these scientific applications execute to completion without requiring further external inputs. Furthermore, the computations performed tend to be rather regular, in that repetitive computations are performed on a vast amount of data. On the other hand, general purpose applications have their execution flow determined by the user (and other external inputs). These applications tend to be more interactive and hence the computations tend to be irregular depending on I/O actions. Consequently, the event-driven processing model is widely used, in which a series of events is handled sequentially. As such, improving the performance for these programs not only includes the computing speed, but also the responsiveness. In order to improve user experience, GUI applications need concurrency for a better responsiveness. This consideration leads to the idea of integrating OpenMP’s traditional parallelism aspect with the concurrency requirements of GUI applications – in a way that conforms to OpenMP philosophy, and also from a software engineering approach to increase its embracing from the wider programming community.

The last salient difference is, there are large amounts of use of advanced data structure and objects in modern software. Conventional batch-like programs directly manipulate primitive data. In the light of large scale software developing, most data in object-oriented programming are highly encapsulated and abstracted. For instance, it is normal to use iterators in the for-loop when programming in Java, where explicitly assigning the start, end and stride of the iteration is unnecessary. The poor support for advanced
iteration and object sharing in OpenMP makes it difficult for the software developed in these object-oriented languages to benefit from parallel speedup.

**Pyjama**

A combination of object-oriented concepts and parallelism can definitely improve both software robustness and efficiency [50]. This is the main focus of Pyjama.

An overview of the Pyjama framework

A glance at the Pyjama framework is shown in Figure 2. In general, the Pyjama framework constitutes the Pyjama compiler and the Pyjama runtime support.

The semantics of Pyjama are essentially a standard Java with the OpenMP extension. Because in Java the `#pragma` macro is not valid, and in order to let standard Java compilers ignore the directives and compile the source code in a normal sequential manner, every Pyjama directive begins with `/#omp`. A program line beginning with `/#omp` is treated as compiler directives by the Pyjama compiler and ignored as inline comments by other Java compilers. The generic syntax is as follows:

`/#omp directive-name [clause [, clause] ...]`

The Pyjama compiler performs a source-to-source translation. The standard Java code (i.e. the code not annotated with Pyjama’s OpenMP directives) is retained without any
changes. All other code blocks annotated with OpenMP directives are transformed into parallel Java code. The process of Pyjama compilation can be divided into four stages as follows:

**Parse and Normalization.** The compiler’s underlying parser is generated using a parser generating tool, JavaCC [51]. This allows the parser to process Java code with OpenMP extensions. When the Java OpenMP source code is processed by the Pyjama compiler, this input stream is generated into an Abstract Syntax Tree (AST) according to the grammar. Meanwhile, during the parsing stage, necessary code normalization is applied which is meant to convert some code snippets into standard representation (e.g. convert all OpenMP sections into a standard OpenMP for-loop).

**Symbol Scoping Visiting.** After the AST is generated, a series of AST traversals is applied by using the Visitor Pattern [52]. This includes a symbol scoping stage where all nodes and variable information are extracted from the AST and stored into a symbol table. This information includes the types of all nodes, the lifespan of every variable and function, variable renaming translations, and so on.

**Parallel Code Translation Visiting.** Since the AST contains OpenMP nodes, this stage transfers the OpenMP code into standard Java code. This is accomplished with the PyjamaToJavaVisitor. The sequential version is translated into the parallel version by incorporating the Pyjama runtime routines. Finally, the parallel Java code is dumped into the output stream from the AST.

**Javac Compiler.** Finally the generated parallel Java code is compiled into the Java bytecode using a standard Javac compiler. The bytecode can be executed in the Java Virtual Machine (JVM), together with bytecode compiled from standard Java code and external libraries. During the parallel execution, the Pyjama runtime library is responsible for the parallel thread management and scheduling.

*Pyjama extended features*

Based on OpenMP Specification 2.5, Pyjama supports a super-set of parallel programming facilitation. Some noticeable aspects are described below.

*GUI-aware mechanism*

In the official OpenMP specification, the master thread is a part of the parallel thread group. For GUI applications, implementing an event handler using OpenMP fails. Since
the Event Dispatch Thread (EDT) is the master thread, when it encounters an OpenMP construct the EDT will be part of the processing team. As a consequence, the GUI application becomes unresponsive.

In order to solve this problem, a preliminary solution has been proposed [53]. This solution endows OpenMP with GUI awareness. The thread model is extended by two new directives, `freeguithread` and `gui`. When the EDT encounters a `freeguithread` construct, a substitution thread is created to do the work on behalf of the EDT. The EDT is then able to skip the execution of the `freeguithread` block, and returns to the event loop. By using the `gui` directive, executions related to the GUI can be posted back to the EDT from a background thread, which avoids synchronization problems with GUI updates.

The combination of `freeguithread` and `gui` directives enables programmers to achieve responsive application development and the threading model still adheres to the OpenMP model.

Listing 1: Pyjama GUI aware directives.

```java
public void actionPerformed(ActionEvent e) {
    //omp freeguithread
    {
        for(File file: list)
        {
            processImage(file);
            done++;
            //omp gui
            progressBar.setValue(done*100/todo);
        }
    } //implicit continuation point
}
```

**Object-oriented loop parallelism**

For Java and other high-level object-oriented languages, high level loops are extensively used. However, conventional OpenMP does not provide high-level loop parallelism such as iterator-based or for-each loops. Pyjama makes it possible for Java to gain the benefit of high-level loop parallelism.
Listing 2: Parallelizing the for-each loop by using Pyjama.

```java
#pragma omp parallel for shared(images)
for(Image img: images)
{
    img.process();
}
```

Listing 3: Parallelize the iterator-based loop by using Pyjama.

```java
#pragma omp parallel for shared(images)
for(Iterator iter=images.iterator(); iter.hasNext();)
{
    Image img = iter.next();
    img.process();
}
```

The `#omp parallel for` directive still conforms with the standard OpenMP specification without any change, but the directive can be used with any Java loop. Listing 2 and Listing 3 show two basic examples of the for-each loop and iterator-based loop quickly parallelized by Pyjama directives.

The implementation of this interface first requires the Pyjama compiler to be able to parse the high-level loops and the information about the loop is extracted. Second, during the code translation, Parallel Iterator [54] is used. Parallel Iterator is a data parallelism solution for object-oriented computations. It can be used to traverse a list of data by multiple threads without data race and contentions.

Custom reductions

Pyjama also provides a high-level data reduction interface. Programmers can define functions as reduction operations, which can be used in Pyjama’s reduction clauses.

RedLib [55] is an extensive library of reduction operations, including nested reductions and reductions for aggregate data types.

Listing 4: Custom reduction using Redlib.

```java
public class Point{
```
private int x, y;
public static void main (String[] argc) {
    Point p = new Point(0,0);
    //#omp parallel reduction(PointReduction:p)
    {
        //parallel region code
    }
}
public Point PointReduction(Point p1, Point p2) {
    // user defined reduction operation here
}

Pyjama Tasking

Task parallelism has been an important part of OpenMP since version 3.0 [56]. OpenMP tasking enables programmers to handle irregular and unsymmetrical parallelism problems that the traditional worksharing constructs could not solve. The evolution of the OpenMP specification has provided increased flexibility and expressiveness with tasks, such as dependency handling [57] and task-generating loops [58].

The current implementation conforms to OpenMP Specification 3.0. The task construct defines an explicit task, and it is only active when this construct block is within a parallel region. One noticeable remark is the clause untied is not implemented, because the Java Virtual Machine (JVM) thread scheduling is delegated to the operating system, and from Java's perspective a task will always be executed by one specific thread.
Directive Syntax

```
//#omp task [clause[,clause]...]

structured-block

clause:

data-handling-clause

if-clause

where data-handling-clause is one of the following:

firstprivate(list) shared(list)

and if-clause is:

if(scalar-expression)
```

Figure 3: The task directive in Pyjama.

Implementation

The implementation of OpenMP task is composed of two parts. First the source-to-source compiler that transforms the OpenMP task code block into a Java class with parallel execution property. Second the runtime support provides the underlining threadpool management, task scheduling, and related OpenMP runtime functions.

auxiliary class generation

An auxiliary class (which is an inner class of the current compilation unit) is generated to represent the specific OpenMP construct block. In general, when the compiler processes the sequential code, each task code block is refactored into an inner class, and this inner class inherits an abstract class called OmpTask. The abstract interface call() is implemented to include the user code. Meanwhile, all the variables which are used in the target block are also required as field variables in the auxiliary class, and they should be passed in and initialized by the auxiliary class constructor.

task block invocation

In the generated code, the invocation of every task block is converted to the invocation of its paired auxiliary class. First, an instance of its auxiliary class is initialized, with proper arguments. Second, by checking the current Internal Control Variable (ICV), the runtime detects if the current thread is a member of the OpenMP parallel thread group. If yes, this task will be submitted to the corre-
sponding thread pool and returned to the thread. Otherwise, the task block executes sequentially.

**Compared to C/C++ version of OpenMP task**

The implementation overhead of the Java version is clearly more than that of a C/C++ version. For example, an implementation based on light-weight execution units called nano-threads [59] has been proposed [56]. The nano-thread layer is implemented with POSIX’s pthreads and this layer has a slight impact on efficiency. In contrast, because of the nature of Java, it cannot directly involve any system calls and all operating system level calls have to be delegated to the JVM runtime. In the meantime, the code transformation from the sequential version to the parallel version inevitably introduces new classes and their invocations causing more overhead in the runtime. It is foreseeable that the Java version of OpenMP tasking has a higher threshold than C/C++ to see the real benefit of task parallelization. In another words, the purpose of using Java tasking should mainly target coarse-grained tasks rather than fine-grained tasks.
The proliferation of parallel processing in shared-memory applications has encouraged developing assistant frameworks such as OpenMP. OpenMP has become increasingly prevalent due to the simplicity it offers to elegantly and incrementally introduce parallelism. However, it still lacks some high-level language features that are essential in object-oriented programming. One such mechanism is that of exception handling. In languages such as Java, the concept of exception handling has been an integral aspect to the language since the first release. For OpenMP to be truly embraced within this object-oriented community, essential object-oriented concepts such as exception handling need to be given some attention.

The official OpenMP standard has little specification on error recovery, as the challenges of supporting exception-based error recovery in OpenMP extends to both the semantic specifications and related runtime support. This chapter proposes a systematic mechanism for exception handling with the co-use of OpenMP directives, which is based on a Java implementation of OpenMP. The concept of exception handling with OpenMP directives has been formalized and categorized. Hand in hand with this exception handling proposal, a flexible approach to thread cancellation is also proposed (as an extension to OpenMP directives) that supports this exception handling within parallel execution. The runtime support and its implementation are discussed. The evaluation shows that while there is no prominent overhead introduced, the new approach provides a more elegant coding style which increases parallel development efficiency and software robustness.

The contributions of this chapter can be divided into three parts:
• The categorization and formalization of object-oriented exception handling in OpenMP parallel regions.

• The concept of flexible thread cancellation is proposed, which provides a better approach for managing the control flow of a program, as well as facilitating exception handling on threads.

• Usability and performance are evaluated through an OpenMP implementation for Java.

The contents of this chapter are based on published papers in the Proceedings of the 11th International Workshop on OpenMP [60], and International Journal of Parallel Programming [61].

This chapter is organized as follows. Section 3.1 starts off with the introduction. Section 3.2 reviews the latest studies related to the OpenMP error recovery models, and finds that there is no dedicated error recovery model for object-oriented programming. Section 3.3 gives an overview of the current state of using error recovery in object-oriented languages, and lists the problems programmers are facing. In Section 3.4, an extended parallel cancellation solution is discussed, which will give programmers more flexibility to express parallel control flow and is also useful with alongside exception handling. In Section 3.5, exception handling with OpenMP is comprehensively discussed. Implementation issues are discussed in Section 3.6 and the evaluation of the new runtime support is provided in Section 3.7. Section 3.8 summarizes this chapter.

INTRODUCTION

The OpenMP programming framework still has some way to go before it is widely used for general software development. In particular, the current OpenMP standard lacks support for essential programming features such as mechanisms for error recovery. As a matter of fact, OpenMP is mainly used for compute-intensive applications that are deterministic and less error-prone, such as batch-like, or numerical and scientific computations. For other kinds of parallel programs (such as server-side applications [62, 63], games [64], desktop and mobile platform software [65]), which are typically interaction-based, handling unexpected situations is essential for robustness.

Exception handling is an error recovery mechanism which enables programs to anticipate and recover from abnormal situations and consequently avoid any abrupt ter-
mination of applications. Compared with other error handling approaches (e.g. error code based, callback function based [66]), exception-based recovery is more compliant with object-oriented principles, due to its support for user-defined exceptions. In object-oriented languages, useful information about an error is typically stored in an instance of an Exception class. Moreover, it is lexically clearer and more flexible to directly surround code that could potentially throw exceptions in try-catch-finally blocks. OpenMP does not provide rich support for object-oriented exception handling in parallel environments. If anything, considering that a parallelized application is likely to introduce more potential problems than that in a sequential application, this lack of support for exception handling makes it especially difficult to write robust object-oriented parallel code using the OpenMP approach. This is especially important to recognize in an object-oriented language such as Java, where exception handling is an integral part of the language. As Android and multi-core mobile devices continue their dominance, parallel programming is evermore relevant and presents another opportunity for OpenMP to embrace this community of developers.

A direct combination of the conventional object-oriented exception handling model with OpenMP is not feasible. This is due to the conventional exception handling mechanism being only compatible with single-stream control flow. The catch block works as a backup execution stream and does not execute unless the specified exception occurs inside its paired try block. Moreover, it is guaranteed that at most one exception may occur within the contained try block, since the control flow is executed sequentially. Therefore, either an exception is handled adequately and the program continues running, or the exception is propagated upwards (and may potentially terminate the program if not handled at a higher level). However, this procedure becomes more complicated when developing code in OpenMP, because OpenMP directives change the context of the (otherwise lexically sequential) source code. That is, certain regions of the code might be executed in parallel (i.e. the parallel regions), and multiple exceptions in those parallel regions may need to be handled differently. In other words, exception handling with multiple control flows cannot simply adopt the conventional sequential try-catch-finally policy. Since the program deals with multiple threads, there are several factors that need to be considered:

- Differentiating between handling single-thread exceptions and thread-group exceptions.
- Whether to stop the entire execution if one of the threads encounters a non-handled exception.

- Ensuring that stopping a thread does not interfere with the execution of other threads.

An in-depth examination for exception handling in an OpenMP environment has been proposed and the performance will be evaluated.

RELATED WORK

Although the official OpenMP standard does not have a comprehensive error handling mechanism at the moment, several error handling models have been proposed for OpenMP. Gatlin [66] initially classifies error handling into three categories based on exception, callback function and error-code. Exception-based error handling is widely used in object-oriented languages such as C++ and Java, but combining this mechanism with parallelization approaches in OpenMP has not been studied in depth so far. On the contrary, error recovery models that are based on callback functions are widely used in different domains, but they seem to be too complicated to use. Low level languages such as C and Fortran mainly use this approach to handle errors, as these languages lack proper exception handling mechanisms. For this category, Duran et al. [67] introduces a model for error recovery in OpenMP that is based on callback functions. The model proposes a mechanism for registering callback functions using the onerror clause to specify a function that is called in case of a specific error. Moreover, Wong et al. [68] discussed the necessity of error-handling models in OpenMP. However, they argue that the model must support exception-unaware languages (e.g. C and Fortran), thus their model does not include the semantics of exception throwing and try-catch blocks. Kao [69] discussed the issue of exception handling in C++ OpenMP programs, however, some concerns are only confined in C++. For example, C++ manages memory by pointers and allocated memory needs to be reclaimed manually when handling exceptions. As a comparison, in Java, Java Virtual Machine (JVM) is responsible for garbage collections and its exception handling does not require any explicit memory management. Another example is Java supports finally keyword, by which resource handlers can be safely closed no matter whether exception happens, but in C++, it is not available.
Aside from OpenMP, some efforts have been made to propose the semantics of exception handling in concurrent programmings. For example, Keen and Olsson [70] proposed an exception model that supports handling exceptions thrown from an asynchronously invoked method. Zhang et al. [71] proposed an as-if-serial exception handling mechanism for Java Futures. Parallel Task, another task-based parallelization tool [31], introduced the asyncCatch model, to integrate asynchronous exception handling with object-oriented programming concepts.

PROBLEM OVERVIEW

In this section, the obstacles towards efficient and robust exception handling programming are itemized with the help of some code snippet examples.

Current situation

Although it may be possible to handle exceptions thrown within OpenMP parallel regions, it is rather counter-intuitive, demanding and confusing to correctly implement since the semantics are evaded in the OpenMP standard. According to the specifications of OpenMP 4.5 [32], when an exception is thrown inside a parallel region, the only restriction is that the exception should be caught and handled within the same region and by the same thread. Therefore, a parallel region surrounded by a try-catch block does not comply with OpenMP specifications (see Figure 5). Moreover, it also cannot guarantee that a try-catch block within a parallel region will function as expected, due to some semantic defects within OpenMP specifications [72]. For example, Figure 6 shows a try-catch block embedded inside an OpenMP parallel region. Although this syntax may get through an OpenMP compiler, it has a potential runtime bug. In this particular case, when an exception occurs before the barrier, the control flow of the encountering thread will jump to the catch block. This jump will skip the barrier directive, while the other threads that do not encounter an exception end up halting indefinitely at the barrier synchronization. This is similar to the reason why OpenMP standard strictly follows the Single Entry, Single Exit (SESE) principle as [72] indicated. Although there are already some static analysis techniques proposed such as [73] which is designed for checking the validation of barriers, it still lacks the consideration onto exception handling semantics.
Listing 5: Try-catch mechanism that does not syntactically and semantically conform with the OpenMP specification.

```c
try{
    #pragma omp parallel for
    for(int i=0; i<4; i++){
        cause_exception();
    }
}catch(Exception e){
    //handling exception
}
```

Listing 6: Syntactically conforms with OpenMP specification, but semantically it has a defect.

```c
#pragma omp parallel{
    try{
        phase1_cause_exception();
        #pragma omp barrier
        phase2();
    }catch(Exception e){
        //handling exception
    }
}
```

**Problem definition**

The current situation of using *try-catch* blocks suggests that programmers encounter difficulties due to programming inconveniences and pitfalls of OpenMP error handling. Lacking a standard and consistent error handling mechanism in OpenMP makes programmers struggle in writing robust and efficient OpenMP code. The major consequence of the lack of exception handling mechanisms in OpenMP hinders the widespread use of OpenMP in object-oriented languages, since there is no clear OpenMP conformity with contemporary software design paradigms. Generally, error handling in OpenMP needs to be improved in three major aspects:

- The semantics for checking whether catching an exception can cause other problems.
• Convenient and flexible mechanisms for controlling or canceling execution within parallel environments.

• A reliable runtime support for the default behavior of parallel executions when they encounter uncaught exceptions.

CANCELLATIONS

Before discussing exception handling within OpenMP parallel regions, it is helpful to discuss the significance of cancellation in a parallel context. In sequential programming, canceling execution at a certain part in the code is easily achieved by using the supported programming language keywords (e.g., break to cancel a loop, or return to cancel execution within a method). Because there is only one control flow, cancellation in sequential code simply means canceling the current scope of execution. In an OpenMP parallel region, such a cancellation keyword is lexically in a sequential program but semantically executing in parallel (the OpenMP philosophy that the original sequential code is intact when the OpenMP compiler directives are ignored). In this parallel context, does a cancellation indicate the termination for a single thread in the parallel region (i.e. the one encountering the cancellation), or would it indicate termination of all threads participating within the current parallel environment? Therefore, when converting sequential code to parallel code, extra directives are needed to convey the programmer’s intentions. This type of directive should be flexible and easy enough to express programming logic, while still respecting the OpenMP approach of maintaining lexically sequential code.

OpenMP 4.0 standard has added some directives related to region cancellation [74]. According to the cancel directive, programmers are allowed to cancel the innermost parallel/for/sections/taskgroup region where the cancel directive appears. This specification provides an approach to stop execution of a parallel region, with the combination of cancellation point directive, which allows for user-defined cancellation points. The net effect of this directive is that it results in stopping the entire parallel execution. The cancel directive lacks the ability to stop a single thread locally without interfering with the execution of other threads. This would be useful when a thread encounters an exception and cannot recover from it, so it may be desirable to only stop execution of that current thread (since it no longer needs to continue its assigned workload), without canceling the entire parallel execution. Using a break-statement goes against OpenMP
standards (since it is oblivious to OpenMP barriers). The status quo makes it difficult for programmers to specify the control flow of a parallel execution, and confines the use of exception handling when exceptions happen in a parallel execution.

**cancellation directive** In order to support a more flexible thread canceling mechanism, and to better support the OpenMP exception handling model, the official cancel directive is extended. This extension is achieved by adding a `thread-affiliate-clause`, which can be `global`, indicating the cancellation of the entire thread group (the current OpenMP definition), or `local`, merely indicating the cancellation of the current thread encountering the directive. The optional `if` clause, signaling that the cancellation is active only when the condition inside the `if` statement holds true, remains unchanged. The optional `throw-clause`, indicating an extra exception throwing when the cancellation is applied. Figure 4 demonstrates the extended syntax of the cancel directive. There is an additional optional clause proposed, `neglect_exception`, for constructs `parallel`, `for`, `sections` and `taskgroup`. This is under consideration for the simplification of parallel exception handling, which will be explained in Section 3.5.3.3.

```
#pragma omp cancel construct-type-clause thread-affiliate-clause [if-clause] [throw-clause]
```

where `construct-type-clause` is one of the following:

- `parallel, sections, for, taskgroup`

and `thread-affiliate-clause` is one of the following:

- `global, local`

and `if-clause` is:

```
if(scalar-expression)
```

Figure 4: Extended cancellation directive.

The extended cancel directive expands the control over a group of threads. That is, by combining different clauses, programmers can express customized behaviors of the parallel control flow. Figure 5 visualizes the cancel directive with the combinations of different clauses. Black nodes indicate the cancellation triggering points. A thread with a black node is the cancellation triggering thread. If a thread encounters a cancellation directive with the local property, it will only stop executing the innermost OpenMP construct thread-locally. Afterwards, the thread resumes when all other threads within the parallel execution reach the next statement following the canceled region. On the other hand, if the cancellation is a global cancellation, the triggering thread will set a
Cancellation triggering point
Cancellation checking point
   (a)      (b)      (c)      (d)

Figure 5: Different uses of cancel directive. (a) cancel parallel local Only single thread quits the innermost parallel region; (b) cancel parallel global Entire thread group quits current parallel region; (c) cancel for local Single thread quits current worksharing for-loop, but continues when other threads finish this for-loop iteration; (d) cancel for global All threads quit the current worksharing for-loop and continue with the following statement.

global cancellation flag. Other threads check this cancellation flag at next cancellation checking points (indicated by white nodes). Afterwards, all threads resume from the next statement after the cancellation region.

The cancel directive can be used for two purposes: First, programmers can explicitly use this directive to express parallel control flow. Second, it works as an implicit operation when an exception happens within a parallel execution. The latter is explained in more detail in Section 3.5.3.3.

EXCEPTION HANDLING

This section demonstrates the exception handling model. In order to ensure the robustness and flexibility, several limitations and extensions are discussed. The discussion is categorized into two parts: Local exception handling and global exception handling.

Overview of Categorization

In proposing a comprehensive model for parallel exception handling, this section discusses different categories of exception handling in order to set up a standard for using exception handling with OpenMP directives to prevent unexpected execution behaviors. There are two kinds of exception handling scenarios that would be useful in an
OpenMP environment. One involves handling exceptions within a single thread, while the other involves exception handling across a group of threads:

**Local exception handling:** This means an exception is handled by the same thread that threw the exception in the parallel region. A successful local handling must ensure that the procedure of error recovery does not influence the execution of other threads. A local exception handling *try-catch* block does not surround the entire parallel region, but is rather handled internally within the parallel region.

**Global exception handling:** A global exception means an exception that potentially influences the entire parallel region. If an exception in a parallel region is not caught by its throwing thread, or handling this exception causes another exception to be thrown, then the exception will affect the entire parallel execution. The OpenMP standard does not categorize this behavior, since it insists it should never occur. Lexically, the *try-catch* block for handling of these types of exceptions would surround the parallel region in which the exception might happen. An uncaught global exception will make the entire parallel execution stop. If this exception is still not caught afterwards, the entire program will stop as well.

**Local Exception Handling**

Local exception handling ensures that errors are recovered inside their local threads, and the local threads continue working/progressing. In order to avoid an unexpected execution behavior (examples in Section 3.3.1), this type of handling requires two conditions to be met: (a) Any potential exception inside a *try-catch* block does not interfere with other thread’s execution; (b) Any operation inside a *catch/finally* block does not affect the entire parallel region’s progress.

Technically, as a legal local exception handling, the entire exception handling region requires there is no OpenMP synchronization point presenting, in either of the *try-catch* or *finally* blocks. Furthermore, it should be ensured that (a) there is no exception re-throwing or (b) if exception re-throwing happens, the re-thrown exceptions need to be handled by another legal local handling.

With regard to parallel synchronization points in the parallel region, usually represented by various OpenMP directives, this can be categorized into two groups:

**Control-flow synchronization point:** A control-flow synchronization point is defined as a point where a thread cannot evolve until it is synchronized with other threads in
Table 1: Two types of synchronization point in OpenMP.

<table>
<thead>
<tr>
<th>control-flow synchronization point</th>
<th>context-property switching boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp barrier</td>
<td>omp parallel</td>
</tr>
<tr>
<td>omp for (implicit barrier)</td>
<td>omp section</td>
</tr>
<tr>
<td>omp sections (implicit barrier)</td>
<td>omp single</td>
</tr>
<tr>
<td>omp single (implicit barrier)</td>
<td>omp master</td>
</tr>
</tbody>
</table>

The corresponding parallel region. A typical control-flow synchronization point is the barrier directive. Other directives, may contain an implicit barrier if the nowait clause is not specified. Those directives include for, sections and single. If there is a control-flow synchronization point inside the try block, there is a risk of not being reached by one of the threads when this thread encounters an exception.

**Thread-context switching boundary:** The attribute of source code changes when encountering a thread-context switching boundary. In an OpenMP parallel region, there are mainly three types of source code regions: (a) Code regions to be executed by every thread at the same time; (b) Code regions to be executed only by one specified thread (e.g. master) or non-specified thread (e.g. single); (c) Code regions to be executed by every thread, but the executions need serialization (e.g. critical). A thread-context switching boundary works as a dividing boundary to change this thread-context property. Notice sometimes a control-flow synchronization point is also a thread-context switching boundary, such as for. If a try block contains several OpenMP code blocks which represent different thread-contexts, it is easy to cause an ambiguous exception handling semantic and unexpected runtime behavior. So, avoiding thread-context switching boundaries inside a local exception handling try-catch block is a better programming practice.

**Example** In order to illustrate the concept of parallel synchronization point, Figure 6 shows a visualization of all synchronization points onto a piece of dummy OpenMP source code. All parallel synchronization points slice the code snippet into several pieces, indicated by the lines. A legal local handling, a.k.a try-catch(finally) block, should not traverse any line as indicated. This restriction confines programmers using try-catch
blocks inside certain code span, to ensure the exception handling does not interfere with any other thread’s execution.

According to this limitation, a robust compiler should be able to throw a warning to inform programmers if the OpenMP source code does not conform to local exception handling rules. This warning reminds programmers to double check the code to see whether the exception handling could cause any side effect.

Implementing this feature can be safely achieved by adding an extra semantic checking pass at the front-end parsing stage of compilers. The general idea is, when encountering a `try` block, the parser becomes sensitive to checking certain types of OpenMP directives, and when encountering a directive or an ending border of a directive which has the similar property to a synchronization point, the compiler throws a warning from it.

---

Figure 6: A legal local exception handling `try` block cannot traverse any line as indicated.
Global Exception Handling

Global exception means an exception is emitted from a parallel region and is not handled thread-locally. It indicates an unexpected behavior has occurred and escaped from within the parallel execution. If this exception is not handled by its local thread, this exception will be forwarded to the parallel region. Because a thread-locally-uncought exception could influence the correctness of parallel execution, this exception changes its property and becomes a global exception and handling this type of exception is defined as global exception handling.

Global Exception Catch Procedure

In a sequential program, if an exception happens, it needs to be handled by the encountering thread. If the thread cannot find a matching catch block, the program will stop with throwing an unhandled exception. However, in parallel execution, if an exception happens in a thread, it is not always necessary to stop the parallel execution. Programmers can specify the behavior when an exception happens, that is, to handle it by the encountering thread, to expose it to the parallel environment, or to stop the encountering thread only.

Figure 7 shows the flowchart for the case of an exception within an OpenMP parallel execution. When a parallel program is executing, if it encounters an exception, it first checks whether a local exception handler is defined. If yes, this exception will be handled using the thread-local approach, and then the encountering thread continues processing. Notice that it is possible to throw another exception from the handling code (i.e. catch or finally block), in which case the program continues looking for another local handler until the exception cannot be handled locally. If a thread encounters an exception and this exception is not handled locally, the default behavior will be cancel parallel global, which triggers the cancellation of that parallel region. In another situation a program may encounter a cancel directive. As discussed in Section 3.4, cancel directives can also be used for deliberate control-flow stops. Therefore, execution stops due to OpenMP cancellation are not always regarded as exceptions.

Exception Registration

In some cases, programmers may need to throw an exception when encountering a cancellation directive. Since cancellation is always the last reachable statement in a
Figure 7: Flowchart of exception handling within an OpenMP parallel region.
code scope, directly appending a throw statement does not work. In order to solve this problem, the designed syntax enables cancellation directives to register exceptions. This mechanism allows programmers to define which kind of cancellation inside a parallel region requires error recovery. Furthermore, during runtime, parallel execution only regards the cancellation directives with exception registration as unexpected exits. In order to achieve this behavior, a new throw clause is added to the omp cancel directive, which indicates this cancellation is followed by an exception throwing from the innermost parallel region. The throw clause provides more flexibility for expressing parallel regions, and makes code more readable with cancellation directives that throw exceptions explicitly. The latter is important for the maintenance of parallel source code, from a software engineering point of view.

An example of exception registration is showed in Listing 7. Inside the parallel region, there are two omp cancel parallel global directives. In order to distinguish the difference, the first cancellation directive is appended with an exception throwing, which means regarding the first parallel cancellation as an exception. Then if the first cancellation happens, an exception is thrown out from parallel region and it can be caught by the catch block outside the parallel region. On the contrary, if the second cancellation directive is activated, though the parallel region will be canceled, no exception will be propagated.

Listing 7: An example of using the throw clause.

```java
Array<Object> arr = ...;
Value target = ...;
Object tarObj = null;
try {
    //omp parallel for shared(arr)
    for(int i=0; i<arr.length; i++) {
        if (arr[i] == null) {
            //omp cancel parallel global throw(NullElementException)
        }
        val = process(arr[i]);
        if(val == target){
            //omp atomic write
            tarObj = arr[i];
            //omp cancel parallel global
        }
    }
```
Exception Neglecting

In some cases, it is not desirable to stop the entire parallel processing once an exception is exposed to the parallel environment. Programmers may want the remaining threads to keep executing even if one or more threads fail within the thread group. This can be achieved by explicitly declaring a local cancellation at the end of local exception handling code to make the encountering thread stop locally. However, if there are no other recovery operations within the handling code, the semantic can be simplified by using the `neglect_exception` clause after the corresponding OpenMP construct. When an exception happens and it is registered by the exception neglecting mechanism, it will not trigger the parallel or worksharing execution cancellation. Instead, only the encountering thread will stop. In the meantime, since it is possible that some works distributed by the stopping thread are not finished, for compensation, a dynamic work redistribution is run when the thread stopping happens.

Using an exception neglecting mechanism also enables programmers to easily sustain the continuation of parallel processing when a certain thread inside the thread group fails. Because remedy operations are automatically done by the underlying runtime support, programmers are liberated from the arduous work of converting sequential code to robust parallel code.

Listing 8: A demo code using the `neglect_exception` clause to simplify the recovery procedure when an uncaught exception happens inside a parallel region.

```c
#pragma omp parallel
for(;;){
    //omp single
    requests = collect_requests();
    //omp for neglect_exception(Exception)
    for(int i=0; i<requests.size(); i++){
        try{
            data = process_request(requests[i]);
        }
        catch(DataNonconformityException *e){
            //handle exception
        }
    }
}
```
data = response.NONCONF;
}
response[i] = data;
}

//#omp single
send_responses(responses);

Figure 8 shows a code example which uses the neglect_exception clause to simplify the programming logic. Inside the parallel region, an infinite loop is executed. For each loop iteration, firstly one of the threads inside the thread group collects requests from the network sockets. Afterwards, a series of requests is processed by the thread group. During the worksharing process, exceptions could happen. But only some types of the exception are handled thread-locally. Other unexpected exceptions (e.g. OutOfMemoryException) could still escape from the local thread. Under this circumstance, using the neglect_exception clause followed by a more general exception type (Exception) enables the parallel execution to ignore the exceptions which are exposed to the parallel environment. After an automatic work redistribution (if necessary), the parallel execution keeps processing.

Source Code Simplification

Due to the lack of specifications for parallel exception handling, a conventional traverse-parallel-region exception handling solution would have to use predefined references (or pointers) to store the exceptions that happen in a parallel region. That is, programmers have to manually store exceptions that could possibly occur in a parallel region, and then invoke a global cancellation directive to stop the parallel execution when handling that exception. Thus, a parallel region must be followed by a series of inspections to test whether any of the specified exceptions have happened. The source code (Figure 9) for such a manual approach is quickly tainted with multiple try-catch blocks, especially when programmers want to catch several potential exceptions in a parallel region.

The source code can be easily simplified using new proposed exception handling semantics with OpenMP directives. The try-catch block can directly surround a parallel region without code re-factoring (See Figure 10). The compiler source-to-source generation and runtime support will do all the routines in the background. This improvement
Listing 9: Conventional approach of handling global exception.

```c
ExceptionA * ea = null;
ExceptionB * eb = null;
//#omp parallel shared (ea, eb){
//#omp for
for(int i=0; i<N; i++){  
   try{
      may_cause_ExceptionA();
   }catch(ExceptionA *e){
      //#omp critical 
      {
      ea = e;
      // #omp cancel parallel
      }
   }
}
try{  
   may_cause_ExceptionB();
}catch(ExceptionB *e){
   // #omp atomic write
   eb = e;
   // #omp cancel parallel
}
foo();
} // end of parallel region
if(ea){  // handle exceptionA if happens }
if(eb){  // handle exceptionB if happens }
```
makes the source code more elegant and more compliant with object-oriented design patterns.

Listing 10: Pyjama's new approach of handling a global exception.

```c
try {
#ifdef omp parallel
{
#ifdef omp for
    for(int i=0; i<N; i++){
        may_cause_ExceptionA();
    }
    may_cause_ExceptionB();
    foo();
} //end of parallel region
} catch (ExceptionA *e){
    //handle exceptionA
} catch (ExceptionB *e){
    //handle exceptionB
}
```

IMPLEMENTATION

This section discusses the implementation of enhanced exception handling support. The aforementioned concepts and proposals are implemented through a source-to-source compiler and its runtime support. This section mainly explains some noticeable issues regarding the runtime implementation.

Adaptable synchronization barrier

The extended OpenMP cancellation directive allows the cancellation of a single thread without stopping the entire parallel execution. Since a stopped thread could influence the following synchronization procedure of other remaining threads, it requires an on-the-fly thread consensus number adjustment when a local thread cancellation happens.

This requirement is achieved by implementing an adaptable synchronization barrier. Different from a traditional cyclic barrier, the adaptable barrier has the extra inter-
faces `decreaseConsensus()` and `increaseConsensus()` which enables the barrier to re-adjust the consensus number when a thread quits or joins the thread group. The detailed implementation of the adaptable barrier is listed in Algorithm 3.1. Every thread local cancellation invokes `decreaseConsensus()` before real thread stopping, and the synchronization consensus number decreases from $n$ to $n-1$. The same, if a canceled thread rejoins the thread group, the interface `increaseConsensus()` is invoked and the synchronization consensus number related to this thread group increases from $n$ to $n+1$.

Dynamic work redistribution

As mentioned before, in order to ensure all remaining worksharing chunks are processed if a thread cancels its works in an OpenMP worksharing group, work redistribution is required. This feature can be implemented by different techniques. Wang et al. [75] proposed a very efficient fault tolerant loop scheduling algorithm, which enables the worker threads to re-execute the iterations that should have been executed by the failed threads. Parallel Iterator [76] is another type of thread-safe work distribution technique, and when a thread stops or quits from the parallel execution, the runtime automatically redistributes the remaining iterations to other threads.

The implementation adopts a similar technique as Parallel Iterator does. More specifically, every time a thread quits from a worksharing execution, all its remaining allocated iterations are released and they are collected into a reclaimed pool. If there are still other threads working, then they share these remaining iterations from the reclaimed pool (after those threads complete their normal iterations) using a dynamic schedule with chunk size 1.

If multiple threads attempt to exit from a local cancellation, then all of them will succeed except the last thread, because if there is only one thread, the worksharing construct is at risk of remaining unfinished. Under this circumstance, if the last thread cancels from the parallel execution, an extra exception is thrown out to indicate the total fail of parallel execution.
Algorithm 3.1 Adaptable synchronization barrier.

1: \texttt{consensus}: the consensus number of the thread group
2: \texttt{waitingCount}: initialized as \texttt{consensus}
3: \texttt{lock}: a lock that controls the synchronization
4: \texttt{round}: the condition bound to \texttt{lock}

5: \textbf{procedure} \texttt{doWait}
6: \hspace{1em} \texttt{lock.lock()}
7: \hspace{1em} \texttt{waitingCount} ⇐ \texttt{waitingCount} − 1
8: \hspace{1em} \textbf{if} \texttt{waitingCount} = 0 \textbf{then}
9: \hspace{2em} \textbf{nextRound}
10: \hspace{1em} \texttt{lock.unlock()}
11: \hspace{1em} \textbf{return}
12: \hspace{1em} \textbf{end if}
13: \hspace{1em} \textbf{while} true \textbf{do}
14: \hspace{2em} \texttt{round.await()}
15: \hspace{1em} \textbf{end while}
16: \hspace{1em} \texttt{lock.unlock()}
17: \textbf{end procedure}

18: \textbf{procedure} \texttt{decreaseConsensus}
19: \hspace{1em} \texttt{lock.lock()}
20: \hspace{1em} \texttt{consensus} ⇐ \texttt{consensus} − 1
21: \hspace{1em} \texttt{waitingCount} ⇐ \texttt{waitingCount} − 1
22: \hspace{1em} \textbf{if} \texttt{waitingCount} = 0 \textbf{then}
23: \hspace{2em} \textbf{nextRound}
24: \hspace{1em} \textbf{end if}
25: \hspace{1em} \texttt{lock.unlock()}
26: \textbf{end procedure}

27: \textbf{procedure} \texttt{nextRound}
28: \hspace{1em} \texttt{round.signalAll()}
29: \hspace{1em} \texttt{waitingCount} ⇐ \texttt{consensus}
30: \textbf{end procedure}
**Exception from synchronization regions**

It is possible that an exception is thrown from a critical region. A legal local-exception handling could be available to catch it, as long as it does not break the rule as Section 3.5.2 discussed otherwise, this exception exposes to the parallel region. In order to avoid a deadlock, from the implementation level, it should release the lock resource when the exception escapes from the critical region. In the Java implementation, a `finally` block is sufficient to ensure this, which always makes the lock be released when quitting the critical region. Whereas considering the implementation for C++, which does not support `finally` keyword, RAII [77] is the suitable technique to ensure the life cycle of lock resources is confined inside a certain lexical scope.

**Global exception throwing**

Different from exception handling in sequential execution, in a parallel environment, two or more exceptions may happen at the same time. If those exceptions are not caught thread-locally, then multiple exceptions are exposed to the parallel region. Consider one global exception is thrown from one thread, but before other threads reach the nearest cancellation points, another global exception happens from another thread. If there is no consensus about which global exception should be handled, the entire parallel environment is at risk of inconsistency and unexpected behavior may happen.

In order to ensure exception handling consistency, it is important to guarantee that when multiple global exceptions happen, all the exception exposures to the parallel region should be linear [78] and immediately visible to any other threads within the thread group. This is implemented by endowing each parallel region an exception slot on which the data can be modified using the compare-and-set (CAS) operation. If more than one thread throws exceptions at the same time, the CAS operation ensures that only one exception is set to the exception slot. All other threads which failed putting the exception will put their exception to a logger `ExceptionLogger`. The thread which succeeds on the CAS operation will trigger the cancellation flag and all other threads which fail to register their exceptions will only end with thread cancellations. Algorithm 3.2 shows how a cancellation signal is processed in the runtime.
Algorithm 3.2 The cancellation signal processing in runtime.

1: exception: the encountered in current thread
2: cancellationFlag: the flag that notices canceling parallel execution
3: switch cancellationSignal do
4: case GlobalException
5: if exception is neglected then
6: return
7: else
8: if compareAndSet(null, exception) = false then
9: ExceptionLogger.add(exception)
10: end if
11: end if
12: case GlobalCancellation
13: cancellationFlag ⇐ true
14: case ThreadLocalCancellation
15: barrier.decreaseConsensus
16: cancelCurrentThread

Parallel runtime exception

Parallel runtime exception indicates the potential runtime unexpected behaviors on parallel processing. Since this kind of exception is provided by the parallel runtime library, the exception throwing cannot be reflected from the user source code. Global exception handling enables programmers to handle this type of exception by surrounding parallel region using a try-catch block. Because all these exceptions only happen at runtime, they are all runtime exceptions (inherent class RuntimeException in Java). Sometimes, this type of exception may be fatal for the parallel execution (e.g. OmpParallelStartFailException). In some situations, the parallel runtime exception may influence the correctness of parallel execution (e.g. OmpNotEnoughThreadsException). Programmers can optionally handle these exceptions according to their code purposes. Table 2 lists several selected OpenMP runtime exceptions.
<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Description</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>OmpParallelStartFailException</td>
<td>Failed to start a parallel execution</td>
<td>Fatal</td>
</tr>
<tr>
<td>OmpNotEnoughMemoryException</td>
<td>Does not have enough memory when spawning threads</td>
<td>Severe</td>
</tr>
<tr>
<td>OmpBrokenBarrierException</td>
<td>OpenMP barrier broken due to hardware interruption</td>
<td>Severe</td>
</tr>
<tr>
<td>OmpNotEnoughThreadsException</td>
<td>Cannot generate enough threads as expected</td>
<td>Medium</td>
</tr>
<tr>
<td>OmpSynchronizationTimeoutException</td>
<td>OpenMP synchronization time window runs out</td>
<td>Medium</td>
</tr>
<tr>
<td>OmpUnsupportedFeatureException</td>
<td>Encounter an unsupported feature in current OpenMP version</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 2: OpenMP runtime exceptions.

**EVALUATION**

This section evaluates the new exception handling mechanism in the new OpenMP version for Java. The evaluation mainly contains two parts: The first part describes programmability, which shows how programming productivity has been improved by minimizing lines of code (LoC). Software robustness is also improved by introducing the proposed model. Second, the performance is evaluated by running a series of benchmarks, showing that there is no salient performance degradation with new runtime support, compared with the original unmodified one.

**Programmability**

According to discussed concepts, programmers can benefit from a robust OpenMP program both from the programming stage and the runtime stage, and the guidelines for using this mechanism are as follows:

- First, the compiler does a semantic check to see whether programmers made a legal local thread exception handling in a parallel region. This can prevent unexpected bugs such as the example shown in Listing 6.

- If an exception happens inside the parallel region, and this exception is not fatal, the programmer can use a local cancellation directive to terminate the encountered thread only, keeping the parallel processing alive.
• If the programmer is aware that an exception should be thrown from the parallel region, and the entire parallel execution should be terminated, the global cancellation directive is the best choice to use. Programmers can also register a customized exception that could be handled after the parallel execution termination, by using the `throw` clause.

• Furthermore, if the programmer is aware that an exception should be neglected during the parallel execution, the `neglect_exception` clause can be used to register this exception at a specified region. Only the encountering thread stops if the declared exception happens while the parallel processing continues.

• Last, the runtime support ensures that even if an exception is not handled manually inside the parallel region, the execution will stop the entire parallel execution instead of causing a deadlock (example shown in Listing 5).

Generally, since the compiler and its runtime help do most of the correctness checking and background operations, programmers are able to write robust parallel code with less coding (Listing 10) and effort.

Performance

The overhead has always been the main concern in concurrent and parallel execution. Even though the new design decreases the developing time, having too much overhead introduced during runtime is undesirable, and it defeats the potential benefits of the design. Therefore, the performance measurement of the new design answers two questions:

• Does the parallel `try-catch` block exert a performance penalty when no exceptions happen during runtime?

• Does the new implementation introduce prominent overhead with respect to the conventional manual exception-checking in OpenMP?

The experiments were mainly performed on three systems. The first two are portable multi-core systems, with different infrastructure specifications: The first one has an i5-3570 quad-core Intel processor, with 8M cache, up to 3.90 GHz clock rate, and it would use the HotSpot 64-Bit Server JVM. The second portable system is equipped with a 1.90GHz quad-core Intel CPU i7-3517U and was running the OpenJDK 64-Bit Server
This section mainly focuses on whether the exception handling support degrades the performance even though no exception happens during the parallel execution. A possible overhead can arise from two aspects: (a) either the try-catch guarding on the parallel region, or (b) the explicit cancellation checking points the programmer added into the parallel region.

Experiment 1- Absolute overhead

This experiment mainly evaluates the absolute overhead of launching a parallel execution with a default thread number, with different exception guarding onto the parallel region. Three different coding styles of Normal, Conventional, and New are studied in the experiment. The Normal style involves running a parallel region without any try-catch blocks. The Conventional style includes the traditional approach to handling exceptions, by registering them inside parallel regions and handling them outside parallel regions. The third style (New), is the proposed approach in which a try-catch block is able to directly surround a parallel region and catch any exceptions inside that region. In order to get reliable results, each

![Figure 8: Pure overhead comparison in two systems.](image)

JVM. For simplification, the two portable systems are named System A and System B respectively. The last system is a more scalable system, which is a dedicated 16-core 2.4GHz SMP machine (4 quad-core Intel Xeons E7340) with 64 GB memory, and the Java HotSpot 64-Bit Server VM is used. This system is named System C.

The overhead of try-catch guarding onto parallel region

This section mainly focuses on whether the exception handling support degrades the performance even though no exception happens during the parallel execution. A possible overhead can arise from two aspects: (a) either the try-catch guarding on the parallel region, or (b) the explicit cancellation checking points the programmer added into the parallel region.
The execution time (ms) for different styles in two systems is presented in Table 3.

<table>
<thead>
<tr>
<th>Style</th>
<th>System A</th>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Median</td>
</tr>
<tr>
<td>Normal</td>
<td>0.1826</td>
<td>0.2576</td>
</tr>
<tr>
<td>Conventional</td>
<td>0.1836</td>
<td>0.2737</td>
</tr>
<tr>
<td>New</td>
<td>0.1715</td>
<td>0.2568</td>
</tr>
</tbody>
</table>

Table 3: Key values of overhead measurement in two systems.

A style was run 100 times at one single test, and there were 10 groups of tests conducted in total. Between every two tests, there were random delay periods, in order to simulate a real-time scheduling of an operating system; therefore there are 1000 (100x10) samples collected for each style in each system.

The box plots in Figure 8 demonstrate the graphical dispersion of the samples. Despite 20-30 outliers in each category, most of the runtime samples (970-980) are confined to a narrow interval, and the box regions are very similar to each other. More detailed key values are listed in Table 3. The results demonstrated by the data confirm that there is no obvious overhead deviation between the three different approaches.

**Experiment 2 - How try-catch guarding overhead influences scaling**

The second experiment was performed on a more scalable system (System C). This system is mainly used to evaluate how try-catch guarding influences the performance when increasing the number of parallel processing units.

Because there is no pre-existing OpenMP overhead benchmark suite for Java, according to the EPCC benchmarks [79], this experiment develops similar benchmarks to measure the OpenMP synchronization overheads of the Java version. In the benchmarks, the parallelization overhead is defined as $T_p - T_s / p$, where $T_p$ is the parallel execution time on $p$ processors and $T_s$ indicates the sequential execution time of the same program with the same working load. In order to achieve a consistent and more accurate evaluation on the JVM, each benchmark case was run $n$ (varies between different cases) times and before each benchmark case an $n/10$ times warmup is executed. Figure 9 illustrates the absolute time of synchronization overhead of `parallel`, `for`, and `barrier` respectively, before introducing the support of exception handling.

After the implementation of the aforementioned exception handling support, two types of execution time were measured. The first is the parallel execution guarded
with a try-catch block (TC). The second, in addition to the try-catch guarding, an extra cancellation checking point (CCP) is added. As a reference, the execution time without any exception handling is regarded as the baseline and the overhead differences are computed against it. Figure 10 depicts the absolute execution times of the EPCC-like Java benchmark cases onto parallel, for and barrier that are measured, adopting different coding styles (no try-catch, using TC and using TC&CCP respectively). It can be seen from the diagrams that the execution times do not fluctuate too much compared with the no try-catch guarded one.

In Figure 11, compared with the overhead of no try-catch guarding one, the overheads deviation of TC and TC&CCP are depicted. It can be noticed that the overall average overhead is around 0.15% and the worst case happens with barrier directive on TC.
where the overhead is 3.65% higher than the non-exception-handling one. However, in many of the cases, the overhead is negative which means the execution time of TC or TC&CCP is faster. This phenomenon may be attributed to the operating system scheduling which has a much greater impact on the execution time, so the overhead of exception-handling support does not introduce a noticeable impact on execution time.

The two experiments confirm that the implementation of new runtime does not introduce noticeable overhead in order to support a safer and cleaner semantic.

**Overhead of Global Exception Handling**

In this section the running time of a successful global exception handling is measured. When an exception happens inside a parallel region and it is not handled locally, the thread that encounters the exception should stop, and the cancellation signal for the parallel region is triggered. Moreover, other threads stop when receiving this signal as well. Then, the control flow jumps to the handling code (if any) which is outside the parallel region. The overhead means the time span from the throwing of the global exception to the stopping of the entire parallel execution. The experiment measured the runtimes in this scenario for the Conventional and New approaches. Similarly, the tests were executed 100x10 times for each approach in System A and System B. In every execution, a global exception is thrown and then caught. The average overhead times are demonstrated in Figure 12. The differences suggest that the new proposed runtime support has better performance than the conventional manual approach.

The reason why the Conventional approach is slower is that in order to prevent data race, the exception throwing should be synchronized in a parallel region. the Conventional approach uses a critical region which is lock-guarded. Instead, the new
approach does a code refactoring and the throwing of the exception is a CAS operation. Therefore, obtaining and releasing the lock spends much more time than non-lock-guarded operations. The contention could be very high when many threads throw exceptions to the parallel region at the same time. In contrast, the new implementation uses source-to-source code conversion and all exceptions are assigned atomically, which is faster than lock guarded operations.

Figure 13 depicts the situation with System C. When a global exception happens inside a normal parallel region (parallel construct), the new approach of quitting of the parallel execution is always better than the Conventional approach. However, when it comes to a worksharing region (parallel for construct), it can be noticed that the new approach is slower than the conventional approach for most cases. The explanation
is that the CAS operation is quicker than the lock-based operation, therefore other non-
exception-encountering threads have little chance to get the parallel cancellation signal
at the nearest cancellation checking point and all these threads are still processing until
the next cancellation checking point. Since the parallel execution cannot stop unless
every thread stops, this situation slows down the entire stop of the parallel execution.
This reveals the fact that the position of the cancellation checking points could influence
the stopping time of a parallel region.

**SUMMARY**

The ability to use exception handling mechanisms in OpenMP would be a powerful
feature from a software engineering point of view. The OpenMP specification lacks the
integration of exception handling in object-oriented languages. In this chapter, a combi-
nation of exception handling and parallel programming (based on OpenMP directives)
was discussed. A proposal on the semantics, and the runtime to support these semantics,
is discussed. Programmers will gain a better programming experience when writing ro-
bust high-level parallel code with OpenMP. Evaluations suggest that the new approach
provides an elegant exception handling mechanism in OpenMP, without causing any
performance degradation. A limitation of the proposed solution is that it cannot be di-
rectly applied on the existing high-performance systems that are already implemented
by C/C++ or Fortran. Further more, if this solution is implemented by C/C++, some
extra technical issues need to be considered such as memory management and resource
handling.
Supporting Asynchronization in OpenMP for Event-Driven Programming

The event-driven programming pattern is pervasive in a wide range of modern software applications. Unfortunately, it is not easy to achieve good performance and responsiveness when developing event-driven applications. Traditional approaches require a great amount of programmer effort to restructure and refactor code, to achieve the performance speedup from parallelism and asynchronization. Not only does this restructuring require a lot of development time, it also makes the code harder to debug and understand. This thesis proposes an asynchronous programming model based on the philosophy of OpenMP, which does not require code restructuring of the original sequential code. This asynchronous programming model is complementary to the existing OpenMP fork-join model. The coexistence of the two models has potential to decrease development time for parallel event-driven programs, since it avoids major code refactoring. In addition to its programming simplicity, evaluations show that this approach achieves good performance improvements consistent with more traditional event-driven parallelization.

The main contributions of this chapter are as follows:

- It proposes a simple but expressive virtual target programming model for event-driven programming. The integration of this model with the traditional fork-join model enables for a wider range of applications for OpenMP.

- The asynchronous execution pattern is introduced in the spirit of OpenMP, to overcome the hassles associated with code restructuring. The concept of \textit{async function} is presented to give a function asynchronous execution in nature.
The contents of this chapter are based on the published papers in the 2016 45th International Conference on Parallel Processing Workshops [80], and the Journal of Parallel Programming [81].

The remainder of this chapter is structured as follows: Section 4.1 introduces why supporting asynchronization in OpenMP is of high demand for the event-driven programming model. Section 4.2 reviews the background of developing high-performance event-driven applications, and the difficulties are discussed especially for the development of GUI applications. Section 4.3 presents the proposed programming model, as an extension of OpenMP, and expatiates how this model can help programmers develop responsive event handlers in an efficient way. In Section 4.4, a discussion of the implementation of the compiler and its runtime is provided. Section 4.5 shows the evaluation of this proposed approach. Section 4.6 discusses the related work and Section 4.7 concludes.

INTRODUCTION

Even though OpenMP programming has been widely used in different types of high-performance computing, there are still some barriers which make OpenMP not very suitable for an increasingly essential class of software development: the development of interactive desktop applications and mobile apps. As multi-core devices have become commonplace for the average consumer, especially in the era of ubiquitous computing, it is reasonable to draw the attention of the parallel programming model to the development of everyday applications. Achieving this will allow a larger subset of software apps to really experience the benefit of parallel execution on multi-core devices.

With the interactive nature of these desktop applications and mobile apps, the program flow is executed according to events generated during runtime, known as the event-driven model. Event-driven frameworks are examples of inversion of control [82], which assist developers by only requiring them to be responsible for implementing the event handlers (or callback functions). Although there are various frameworks that differ regarding their implementing languages and supported platforms, the underlying mechanism is very similar. From the programmer’s perspective, they do not need to understand the underlying runtime and its event dispatching, therefore the core part of the application development is implementing the handling routines to reach the required functionality of the application.
In an event-driven application, an event dispatching thread (EDT) is solely responsible for driving the event-loop. Once the application is launched, the runtime support listens for events generated, and queues the event if it is bound to any handling code or callback function the developer has implemented. The callback function is then executed by the EDT. If a particular event handling callback function is time-consuming, the EDT will not be able to handle another event in the event-loop until it finishes execution of the callback function. A problem emerges when the callback function is CPU-intensive or I/O-bound, with the long execution time of the callback function affecting responsiveness of the overall application. For batch-like programs, the motivation for using parallelization techniques is always to decrease the wall clock time. But when it comes to the event-driven programs, performance is not only evaluated by the reduction of the wall clock time. Instead of focusing on execution speedup, maintaining a better responsiveness (and therefore positive user experience) is the main reason programmers incorporate concurrency. For GUI applications, the parallel rendering of the user interfaces provides a better using experience in respect of Human Computer Interaction (HCI) [83].

Due to its focus of accelerating compute-intensive and batch-like programs, OpenMP mainly stresses on the parallelization of loops and symmetrical data processing. Under this consideration, the fork-join model has always been intimately infused into OpenMP, and continues to remain strongly integrated [32]. The fork-join model works well for batch programs and CPU-intensive computations; when the program is launched, its execution rarely interacts with I/O. This is because the workload and work flow are largely pre-defined, allowing for easier reasoning regarding the work distribution. Unfortunately, there are key drawbacks in the traditional OpenMP fork-join model making it incompatible with the co-use of the event-driven programming model.

By its nature, all callback functions are executed by the EDT when the binding event is generated in the event-handling framework. The first challenge facing programmers is conceptually justifying whether a particular computation should be classified as a parallelization candidate. Traditionally, for batch-like programs, programmers would rarely consider parallelizing computations that last only a few seconds. But with interactive event-driven applications, even computations lasting only a few hundred milliseconds demand concurrency to avoid the appearance of an unresponsive application. For OpenMP to be embraced for these mainstream applications, the introduction
of additional overhead for the concurrency of shorter computational spurts needs to be less of a dilemma for programmers.

Regardless of the overhead, the fork-join model presents a much more fundamental issue for event-driven applications. Even with the potential speedup benefits, the traditional fork-join model forces the master thread (which would be the EDT in event-driven applications) to participate in the work-sharing region. This immediately goes against the policies of event-handling frameworks, as the EDT spends a noticeable amount of time away from the event-loop (thereby delaying responses for subsequent events in the application). The traditional way event-handling applications solve this responsiveness problem is by explicitly offloading the time-consuming execution to background threads and then enabling the EDT to return to the event loop to handle another event. While this has long been the standard practice in the realm of event-based applications, it is deprived of the elegance of OpenMP, particularly the paradigm of incremental parallelization that avoids major code restructuring.

Initially, it may appear that OpenMP presents an asynchronous solution with its task directive. However, a block surrounded by a task directive will be asynchronously executed by the OpenMP thread group; an orphaned task directive will execute sequentially unless it is surrounded by a parallel directive. This means the effectiveness of OpenMP tasks are confined within an OpenMP parallel region, conforming to the fork-join model that OpenMP adopts. Since the parallel directive does not provide any option to achieve asynchronization with the parallel region (for example, there is no nowait or async clause), this means that the main thread is forced to wait until every thread in the parallel team finishes its work. This inherently synchronous “join” aspect of OpenMP makes it difficult to integrate OpenMP with the event-driven programming paradigm.

Upon this event-driven programming background, as a clarification, this thesis defines synchronous if every event handler is directly processed by the EDT sequentially. Asynchronous is defined as the event handling being offloaded as a task from the EDT to a background thread, but the task is done sequentially. Parallelization is distinguished from asynchronous, and refers to the execution of a handler with multiple background threads. In synchronous parallel, multiple worker threads are utilized, but this parallelization is foregrounded with the EDT assuming the role of master thread. In comparison, asynchronous parallel means that the event handling code is offloaded to the background,
and then executed in parallel. In this regard, the EDT does not participate in the parallelization.

This chapter first formally defines the cumbersome, yet necessary, restructuring that is demanded to achieve concurrency in an event-driven application. Then a simple but expressive programming model is proposed for asynchronous programming, especially for event-driven programming. An asynchronous executor model is introduced in the spirit of OpenMP, to overcome the hassles associated with code restructuring. Using this model simplifies, as well as unifies, the parallelization and concurrency of event-driven applications. The integration of this model with the traditional fork-join model enables a wider range of target applications for OpenMP. This allows applications that require both asynchronous execution (for event-handling responsiveness) and parallel acceleration (for reduced computational times) to seriously consider OpenMP. The semantic design pattern strictly follows the philosophy of OpenMP, in which adding directives does not influence the original correctness of the sequential execution.

BACKGROUND

This section mainly investigates the background of event-driven programming approaches, and also discusses the difficulties and challenges of developing high-performance event-based applications.

Event-driven programming

A wide range of applications are written based on the event-driven programming model, from desktop and mobile applications (apps) to web services. Different from traditional batch-type programs, event-driven applications do not have a predefined runtime execution sequence. For batch-type programs, given input data, the computations are generally executed until completion without requiring further input from the user. Furthermore, the computations performed tend to be rather regular, in that repetitive computations are performed on a vast amount of data (ideal candidate for parallelization). On the contrary, execution of an event-driven application is achieved by an infinite loop (known as the event-loop) with associated event listeners. When a registered event happens, the listener triggers the callback function implemented by programmers. Due to this major difference, “performance” can mean something
different to batch-like programs than it does to event-driven programs. Batch-like programs mainly stress on absolute execution time, requiring computations be completed as quickly as possible. For event-driven programs, responsiveness (perceived performance) of the application’s interactivity is a major key factor when evaluating its usability ([84, 85, 86, 87]). In Figure 14(i), each triangle represents an event request and the execution of its callback function is represented as a rectangular box with the same color the triangle has. The commencement of request2 is delayed until the handling of previous events are completed, resulting in an unresponsive application.

In order to achieve a good event-dispatching performance and a better user-experience with regards to responsiveness, various solutions exist. The most traditional approach is known as thread-per-request [88]. In this approach, the time-consuming event handling is directly delegated to a newly-spawned background thread. This allows the EDT to directly exit from the event handler, enabling it to handle another event request, hence achieving the desired responsiveness. The first drawback of this traditional approach is the heightened software development experience demanded to effectively multi-thread. There is also the salient drawback of non-scalability, since excessively creating threads could decrease the application’s performance, as well as the overall system performance due increased scheduling demands and increased overhead associated with thread context switching [89].

Figure 14(ii) shows an improved solution making use of tasking concepts and thread pools, instead of creating a new thread preemptively for every event-handler. This involves submitting the long-running code as a task to an executor bound to a thread pool that limits the maximum number of concurrent threads. The executor manages the thread number, thus reducing the threading overhead and improving overall performance when a large number of tasks need to be executed. While this approach addresses the overhead concerns associated with the threading model, it still demands strong conceptual understanding and experience from software developers to parallelize their applications. The dominant conceptual challenge underpinning these models is that a task submission to an executor means operations depending on the result of the task are not allowed to be executed until the task is finished. While this dependency can be achieved by using a blocking waiting operation until the task is finished, it defeats the initial purpose of introducing concurrency if the waiting thread is the EDT. Therefore, the accepted practice is to bind a completion handler to the task, such that the continuing operations will be executed asynchronously when the task is finished.
(i) Unresponsive single-threaded event processing

(ii) Responsive multi-threaded event processing

Figure 14: In an event-driven application, the EDT plays the role of the main thread responding to events. An essential requirement is to maximize the idleness of the EDT, so programmers are required to transform single-threaded event processing to multi-threaded event processing to increase the responsiveness of the EDT.

In addition to the challenge discussed above, another restriction imposed on programmers is that graphical user interface (GUI) components are not thread-safe and access is strictly confined to the EDT. Inside the event handling code, programmers need to identify and separate code segments to ensure thread-safety. For example, in most GUI application frameworks, updates to the GUI should only be executed by the EDT. Disrespecting this rule could result in the user interface exhibiting inconsistency or even errors [90]. Consequently, this means that if handling code is submitted to a worker executor, the thread context may still need to be switched intermittently to the EDT for operations related to GUI updates. This requires further event posting to the EDT with binding callback functions for the display of intermediate results progress.

Language-related dependencies

Implementing event-driven applications largely depends on the application’s language and programming framework. The general aim of the application developer would be to achieve the logic shown in Figure 15. Here, a time-consuming computation is offloaded to the background, while progress updates and final notification still need to be executed by the EDT. Figure 16 and Figure 17 show two specific implementations using Java SwingWorker [91] and C# Asynchronous Programming Model (APM) [92] respectively.
Figure 15: An example of event handling logic, where a time-consuming computation involves background components (S1 and S3), with a foreground progress update (S2), before a concluding foreground computation (S4).

Java SwingWorker enables programmers to identify the operations need to be executed as background tasks or foreground updates, by implementing its class interfaces. As a comparison, the programming style of APM is known as Continuation Passing Style (CPS) [93] and all the continuations of the following operations are asynchronously triggered when the previous operations finish. However, the drawback of using CPS (especially for procedural languages) is prominent. The code refactoring required to achieve this functionality requires fragmenting of the original callback function, where the fragmented statements are wrapped by auxiliary functions. As a consequence, even though the flow logic of the ButtonOnClick() callback function is exactly the same in both implementations, the code structures and API required to achieve this are very different.

This diversity makes it difficult for programmers to write uniform and consistent source code. When porting an application from one platform to another platform, although the programming logic remains the same, the code refactoring requires a great amount of work and programming knowledge.

Notation Representation

This section formally defines the restructuring required to achieve concurrency in an event-driven application. The basic components are firstly defined as follows:

- $e \rightarrow F$ is defined as an event handler binding in which every time event $e$ occurs, the callback function $F$ is to be invoked.

- $F(T)$ specifies that the function call is invoked by thread $T$.

- $F := \{S_1, S_2, S_3, ..., S_n\}$ represents the expansion of function $F$ to represent a total of $n$ statements in that function.
void ButtonOnClick() {
    SwingWorker<String, Integer> worker =
        new SwingWorker<String, Integer>(){
            protected String doInBackground(){
                // S1
                publish();
                // S3
            }
            protected void process(List<Integer> updates){
                // S2
            }
            protected void done() {
                // S4
            }
        }
    worker.execute();
}

Figure 16: Java asynchronous programming with SwingWorker.

- Each statement $S$ can be a primitive statement $P$ or another function call $F$, namely $S ::= P|F$.

- $S(T)$ indicates that statement $S$ needs to be executed by thread $T$.

If there is no concurrency expressed within a function call $F(T)$, all the statements inside $F$ are executed by thread $T$. In this regards, the total execution time of this function is the sum of the execution of each statement: $t(F(T)) = t(S_1(T)) + t(S_2(T)) + t(S_3(T)) + ... + t(S_n(T))$. If the invoking thread $T$ is the EDT, denoted as $T_{edt}$, this means the application is unresponsive for the period of $t(F(T_{edt}))$. As the EDT is executing $F$, it cannot respond to other events or requests during this time.

Regarding to refactoring and asynchronization:

- $F \implies_r F'$ is defined as a code refactoring of function $F$.

- $F|_A$ is defined as an asynchronous execution of function $F$.

For the approach of preemptive multi-threading or task submission, for each $e \rightarrow F(T_{edt})$ there is a refactoring $F(T_{edt}) ::= \{S_1, S_2, S_3, ..., S_n\} \implies_r F'(T_b)|_A$ such that $F'(T_b) ::= \{S_1, S_2, S_3, ..., S_n\}$. Here, $T_b$ is a background thread that has been delegated all statements of the original function. Ideally the handling time of the EDT is decreased to
public class AsyncWorker{
    public IAsyncResult BeginS1(){
        // S1
    }
    public IAsyncResult BeginS3(){
        // S3
    }
}

void S1CallBack(IAsyncResult result) {
    Dispatcher.BeginInvoke(()=>{
        // S2
        worker.BeginS3(S3CallBack);
    });
}

void S3CallBack(IAsyncResult result) {
    Dispatcher.BeginInvoke(()=>{
        // S4
    });
}

void ButtonOnClick() {
    AsyncWorker worker = new AsyncWorker();
    worker.BeginS1(S1CallBack, result);
}

Figure 17: C# AMP-style programming.
zero \( t(F(T_{edt})) = 0 \), allowing the EDT more free time to handle other events. However, the refactoring is typically more complicated as some statements are thread-affiliated.

For example, the formal definition of Figure 15’s callback function can be represented as \( F(T_{edt}) ::= \{S_1, S_2(T_{edt}), S_3, S_4(T_{edt})\} \), where statements \( S_2 \) and \( S_4 \) must only be executed by the EDT. For this situation, a correct and efficient computing offloading of \( S_1 \) and \( S_3 \) could lead to a great amount of code refactoring and programming effort:

\[
F ::= \{S_1, S_2(T_{edt}), S_3, S_4(T_{edt})\} \implies F' ::= \{e_{complete}(F_1(T_b)) \rightarrow F_2(T_{edt}), F_1(T_b)\} \quad (p1)
\]

\[
F_2 ::= \{S_2, e_{complete}(F_3(T_b)) \rightarrow F_4(T_{edt}), F_3(T_b)\} \quad (p2)
\]

\[
F_1 ::= \{S_1\}, F_3 ::= \{S_3\}, F_4 ::= \{S_4\} \quad (p3)
\]

Firstly, the original callback function above is refactored such that \( S_1 \) becomes an asynchronous call by wrapping it with a newly created auxiliary function \( F_1 \), executed by \( T_b \). The completion of \( S_1 \) is then bound to the asynchronous invocation of \( S_2 \), which needs to be executed by the EDT \((p1)\). Similarly, after finishing \( S_2 \), the completion of \( F_3 \) is bound to the invocation of \( F_4 \), while executing \( F_3 \) asynchronously \((p2)\).

This approach successfully decreases the handling time of the EDT to \( t(F(T_{edt})) = t(S_2(T_{edt})) + t(S_4(T_{edt})) \). However, the code refactoring required to achieve this functionality requires fragmenting the original callback function, where the fragmented statements are wrapped by auxiliary functions \((F_1 \sim F_4)\).

This example illustrates that a high-performance event-driven handler implementation is difficult, especially with the combination of parallelization and asynchronization. Besides the creation of thread pool executors and submitting tasks to the appropriate executors, excessive callback function binding, nesting and efforts for the thread executing switching make the logic of the control flow obscure. This traditional approach makes it difficult to get clean and maintainable code for multi-threaded event-based programs. Another salient drawback of using callback function listening is its increased debugging complexity.

**PROGRAMMING MODEL**

The motivation of the semantic design proposed in this section is to provide an OpenMP-like directive-based interface to facilitate event-driven programming. The proposal is
in line with two principles. First, the directive addition conforms with the philosophy of OpenMP, by which the directives can be directly applied on the original sequential version of the code without code restructuring. When the directives are triggered by a supported compiler, the execution benefits from concurrent execution. When the directives are disabled or ignored by unsupported compilers, the code still retains its correctness when executed sequentially. Second, the newly introduced directives are compatible with existing OpenMP directives. With the combination of different directives, programmers are able to express different forms of parallelization and concurrency logic.

**Directive Syntax Extensions**

```
#pragma omp target [clause[,clause]...]  
  structured-block
```

clause:

- target-property-clause
- asynchronous-property-clause
- data-handling-clause
- if-clause

where target-property-clause is one of the following:

```
device(device-number) virtual(name-tag)
```

where asynchronous-property-clause is one of the following:

```
ownait name_as(name-tag) await
```

where data-handling-clause is one of the following:

```
firstprivate(list) shared(list)
```

and if-clause is:

```
if(scalar-expression)
```

Figure 18: Extended target directive.

The proposed syntax (Figure 18) is inspired by the Accelerator Model introduced to the OpenMP 4.0 specification, namely the target directive. The initial purpose of the target directive is to utilize available accelerators in addition to multi-core processors on the system [94]. The target directive offloads the computation of its code block to a
specified accelerator, if a device clause is followed. If the target device is not explicitly specified, the target code block will be submitted to the default accelerator, which is decided by the ICV (Internal Control Variable) default-device-var.

**Virtual target.** The original target directive can only be validated when the host has accelerators (e.g. GPU), which means a valid target must be a physical device. However, the proposed extension of the target syntax introduces the concept of virtual target, by which a target directive can be followed by a virtual clause, instead of a device clause. A virtual target means the computation is not offloaded to a real physical device. Instead, it is a software-level executor capable of offloading the target block from the thread which encounters this target directive. Conventionally, a device target has its own memory and data environment, therefore the data mapping and synchronization are necessary between the host and the target. That is why normally some auxiliary constructs or directives such as target data and target update are used when using target directives. In contrast, a virtual target actually shares the same memory as the host holds, so the data context remains the same when entering the target code block. Figure 19 shows the conceptual differences between the device target and virtual target. Generally, a virtual target is a syntax-level abstraction of a thread pool executor, such that the target block is executed by the executor specified by the target-name.

**Target block scheduling.** By default, an encountering thread may not proceed past the target code block until it is finished by either the device target or virtual target. However, a more flexible and expressive control flow of the encountering thread can be achieved by adopting the asynchronous-property-clause. The consideration behind this is, a target block can also be regarded as a task with an asynchronous nature. Section 4.3.2.2 will specifically explain the different scheduling clauses that influence the processing of the program.

**Semantic Model**

Since the implementation is based on Java, and Java does not support pragma conditional compilation, the directive begins with //omp. This means that compilers that do not support the semantics will safely ignore the directives by regarding them as comments. On the contrary, a supporting compiler will interpret the directives and compile the code as a parallelized version.
(i) Conceptual model of device target.

(ii) Conceptual model of virtual target.

Figure 19: Conceptual difference between virtual target and device target.
void buttonOnClick() {
    Panel.showMsg("Started EDT handling");
    Info info = Panel.collectInput();
    //#omp target virtual(worker) nowait
    {
        int hscode = getHashCode(info);
        downloadAndCompute(hscode);
        //#omp target virtual(edt)
        Panel.showMsg("Finished!");
    }
}
void downloadAndCompute(int hs) {
    Buffer buf = networkDownload(hs);
    Image img = formatConvert(buf);
    //#omp target virtual(edt)
    Panel.displayImg(img);
}

Figure 20: Semantic example of using virtual target directive.

Semantics of target offloading

This section demonstrates the usage of target virtual directives by showing a piece of pseudo code of an event handler implementation. In Figure 20, when a button is clicked, the callback function buttonOnClick() is triggered. Firstly the function updates a message to GUI to indicate the start of the processing. Then a series of time-consuming operations is processed according to the inputs from the GUI. In this example the operations involve downloading a file from the network and then performing image processing on the downloaded raw data. Afterward, the image is rendered to the GUI and a finished message is updated.

If the directives are ignored, the entire code will be executed by the thread which invoked the callback function, i.e. the EDT. For a compliant compiler, the entire callback function will be executed by the cooperation of two virtual target executors (edt, worker). In this situation, the handling time of the EDT decreases because the EDT only spends time on the operations which should be necessarily executed by the EDT. Other operations are smartly offloaded to the worker executor, without breaking the original
code structure and logic. The benefit of using virtual target semantics involves four key aspects:

**Thread-context awareness.** A code block guarded by a specified `target virtual` directive shows its preference of execution by a specified type of thread, or executor. If the encountering thread has the same property as the virtual target specified, the `target virtual` directive is simply ignored. Otherwise, the directive compels the encountering thread to relinquish control of the code block and do a runtime thread-context switch to the specified target. The thread-context awareness property of the `target virtual` construct smartly confines the authorization of the code block execution to the specified type of thread. For example, for GUI applications, a GUI update code block guarded by a `target virtual(edt)` will ensure that all the operations related to the GUI are executed by the event dispatching thread.

**Execution offloading.** Delegating code to another virtual target offloads work from the current thread, therefore alleviating the computational burden from the encountering thread. For a function invocation, if some parts of the function are delegated to other virtual targets, the actual execution time for the thread which invoked the function will be decreased. This aspect is extremely important in the scenario of event dispatching. Work offloading enables the EDT to spend less time on the event handler, allowing it to dispatch more events in the application.

**Data-context sharing.** Using a standard `target` directive means offloading the code to an actual hardware accelerator, therefore data transfer and data synchronization is necessary. Instead, using a virtual target means code is offloaded to a software-level executor. Since all virtual targets share the same memory, there is no need to copy data from the main memory to the accelerators memory. This simplifies usage of the `target virtual` directive, since it is not necessary to do heavy data copying or even variable passing when using a virtual target switch (if the OpenMP `default(shared)` data clause is specified). All the operations inside a target block share the intuitive data context as if the target directive does not exist.

**Intuitive continuation-passing.** Adding `target virtual` directives modifies the source code from a sequential version to an asynchronous (and possibly parallel) version, while still maintaining clean programming logic. The end of a target block is intuitively followed by operations which depend on it. Although a target block has the nature of asynchronous execution when the operations following it should not be executed until the target block is completed, the continuation of the target block does
not require any code refactoring for a completion-event callback function binding. Since the continuation logic is still represented in the sequential code, it dramatically reduces the work of code refactoring to achieve asynchronization and parallelization.

**Semantics of asynchronous execution**

The purpose of a `target virtual(worker)` directive is to offload work from the current thread to a virtual target executor. If the current thread cannot proceed during execution of the target block, and simply halts its execution, there is no actual performance advantage from the target block offloading. Therefore, instead of busy waiting, an asynchronous execution is applied for the target block. The asynchronous execution can be categorized into the three types illustrated in Figure 21, by using different modes for the `asynchronous-property-clause`:

- **Default (wait).** If no `asynchronous-property-clause` is specified, then no asynchronous execution occurs. The encountering thread will busy-wait until the target code block is finished by the specified target. If the executing time of the target block is noticeably long, this is not the ideal approach because the encountering thread cannot do anything useful during this time. Also, in the case of event handling threads, this results in an unresponsive application. However, this wait corresponds to the standard OpenMP behavior of the `target` directive.

- **nowait.** The encountering thread directly skips the target block and leaves the target block as an asynchronous task, then continues executing statements following the block. There is no notification when the task is finished. The `nowait` clause is usually used when there are no further operations which depend on the result of the asynchronous task. Therefore, the code block can be safely invoked and ignored. This is useful for broadcasting interim updates, where the broadcasting thread does not need to wait for a response from listeners.

- **name_as/wait.** The encountering thread directly skips the target block and leaves the target block as an asynchronous task, then continues executing statements following the block. Unlike `nowait`, a task identifier `name-tag` is created that enables the encountering thread to explicitly synchronize with the task by using the associated `wait(name-tag)` clause later in the code. Notice that different target blocks are allowed to share the same `name-tag`, such that when the `wait` clause is ap-
Fig. 21: Different asynchronous modes by using different asynchronous-property-clauses.
plied with that name-tag, the encountering thread suspends until all the name-tag asynchronous target block instances finish.

- **await.** The **await** asynchronization policy is a wait policy, with the important difference that during the wait period the control flow jumps out of the current function and back to its caller. When the target code is finished, the function resumes its execution from where it previously suspended. Conceptually, the purpose of using **await** is, while the target block is being executed by the respective virtual target, the encountering thread is able to return to the event loop (to process other events) or to the task pool (to process other tasks). This is known as Unrelated Handler Processing. This has the advantage of keeping the encountering thread active by processing other meaningful workload instead of blocking or busy waiting. Also, if the encountering thread is the EDT, it enables the EDT to process more events during this time. The code dependency is also naturally represented since the original code sequence is preserved for each event handler; the continuation of an asynchronous execution is intuitive without any need for explicitly binding completion event handlers. Since nested function calls are possible within an event handler, functions may recursively suspend their executions due to an innermost await target block. A more detailed explanation of this situation is discussed in Section 4.3.2.3.

**Semantics of using await and async function**

For most cases, the **await** asynchronization policy is the most convenient way for programmers to express the dependencies between code. Unlike **wait**, which requires programmers to manually guarantee the completion of its corresponding name-tag target code blocks, the code followed by an await target block will be automatically scheduled by the runtime when this await target block finishes.

The **await** property of a target block changes the behavior of the control flow in the current function, since the function yields its execution to the caller when the target block is being executed. As a consequence, the function containing **await** has a resumable (asynchronous) nature itself, and is therefore known as an **async function**. Calling an **async function** is different from calling an synchronous function, because asynchronous functions “return immediately” to the caller before the function’s computation is actually completed (due to the nature of asynchronous execution). Therefore, the caller selects the call type, denoting whether it should (i) invoke the function asynchronously
(and therefore continue on the statement following the function call without waiting for the completion of the asynchronous function), or (ii) also yield its execution and the control flow back to the caller’s caller and resuming only when the asynchronous function completes. Recursively, a function invoking an async function using the latter approach is endowed with a resumable(asynchronous) nature also.

```c
#pragma omp async-call asynchronous-property-clause(function-declaration [,function-declaration]...)
structured-block
```

where asynchronous-property-clause is one of the following:

- **nowait** name_as(name-tag) await

Figure 22: Definition of async-call directive, for the purpose of calling an async function in different ways.

In order to distinguish the two ways to invoke an async function, the async-call construct (Figure 22) is introduced. The directive is followed by an asynchronous-property-clause, then a list of function declarations, and then a code block. All the function calls in the code block that are declared in the function list, will be invoked according to the asynchronous-property-clause. If nowait or name_as is applied, the function is invoked asynchronously and the caller continues with the statements following the function call. The `wait(name-tag)` directive is able to explicitly wait for the completion of its paired name_as(name-tag) asynchronous function call. If await is applied, the caller yields its execution until the declared async function is completed, and then the caller gets the return value from the function and the execution continues.

Figure 23 shows an example of the use of the async-call await construct. In this example, the function `foo()` is marked as an async function because inside this function, a target virtual await block is used. In function `bar1()`, the `foo()` is invoked in a normal way, which means function `bar1()` synchronously invokes `foo()` functions and gets the return values. As a comparison, the function `bar2()` invokes function `foo()` inside an await construct, which causes the suspension of `bar2()` when waiting the return from `foo()` invocations. At the same time, since `bar2()` is endowed as a resumable (asynchronous) nature, when using await construct, function `bar2()` is also marked as an async function.

Formally, a function is defined as an async function when:

1. There are one or more `omp target virtual await` constructs inside this function; and/or
void int bar1(Info info) {
    before();
    a = foo(1) + foo(2);
    after();
}

//#omp async
void int bar2(Info info) {
    before();
    //#omp async-call await(int foo(int))
    {
        a = foo(1) + foo(2);
    }
    after();
}

//#omp async
int foo(int work) {
    //#omp target virtual(worker) await
    {
        cpu_bound_computation(work);
    }
}

Figure 23: Semantic example of using await directive.
There are one or more `omp async-call await` constructs inside the function.

```c
#pragma omp async
function-declaration
```

Figure 24: Definition of async directive, for the purpose of annotating an async function.

An async function should be explicitly annotated, to inform the programmers that this function can be invoked in different ways. This is achieved using `omp async`, to notify the compiler to do the necessary prepossessing (Figure 24). It also informs programmers that the invocation of this function may be different from normal function calls. Invoking this function in the standard way causes it to execute synchronously as would be expected. Instead, if this function is invoked with an async-call await construct, the caller function also becomes an async function.

Figure 25 illustrates the conceptual map of awaiting an async function. ① indicates a caller invokes an async function by using async-call await. This async function is supposed to contain one or more async-call await or target virtual await directives. During the execution of this async function, if there is another async function awaiting or a target block should to be awaited, the execution is delegated by the corresponding executor (can be a specified thread or a thread group), and the caller suspends its execution (②). When the function/target block completes (③), the runtime scheduler enables the caller to resume from the previous suspended point. Then the caller continues its execution (④) with the following execution. This suspend/resume procedure may repeat several times, depending on the number of await directives. Finally the caller gets the return from the async function call (⑤).
**Semantics of using exception handling**

Exception handling is one type of error recovery mechanism which is widely used in high level languages such as Java. By using try-catch blocks, it presents a readable and intuitive control flow since the error-handling code is separated from the normal execution code, and can be categorized according to the type of exceptions/errors. However, the semantics of try-catch is designed for the synchronous execution model: when an exception occurs inside a try block, the thread encountering the exception immediately checks for a corresponding catch block. If it exists, the encountering thread executes the error-handling code and then resumes to the normal execution flow. Otherwise, if there is no appropriate handler found, it propagates this exception to its caller.

When using target virtual blocks, problems may emerge. First, a target virtual directive may change the thread which executes the following block; the thread that encounters the exception could be different from the thread that handles the exception. Second, if asynchronization is applied, the execution of a target virtual block may happen in a future time, therefore when an exception happens from the target virtual block, it already loses the control of its surrounded try-catch block. Although the try and catch blocks are placed lexically correct, semantically the handler is too late due to the async nature.

To overcome these problems and provide a clear specification, the semantic rules of using the try-catch block together with target virtual blocks are defined below:

- If the exception is supposed to be handled and recovered within the target block, a try-catch block should be used inside the target virtual block. This does not break the synchronous execution model since the exception handling thread is the same as the exception encountering thread. This rule is consistent with OpenMP’s exception handling requirements [60].

- If an exception escapes from its inner most surrounding target virtual block, the exception will be re-thrown at the nearest synchronization point. A synchronization point is defined as the point which ensures that the execution of the target block is finished. For **await**, the synchronization point is the point directly after the target code block. For **name_as**, the synchronization point is where its paired wait directive is placed. If an uncaught exception occurs during execution of the target block, the runtime support does not throw it immediately. Instead, this exception is stored and will be re-thrown when the synchronization point is met. Notice
the thread which initially threw the exception can be different from the thread which encounters the re-thrown exception, because of the possible thread context changes.

- If a target virtual block uses the nowait asynchronous clause, there is no chance for it to be re-thrown, since there is no way to verify its completion. In this case, all the exceptions escaped from the target block will be automatically handled by the runtime by printing the stack trace, and allowing the program to continue. This is consistent with exception handling in event-handling frameworks such as Java Swing’s EDT.

**Runtime Library Routines**

Parallelism in the traditional OpenMP fork-join model is triggered by a `parallel` directive. The lifecycle of a thread group is strictly confined within a parallel region, so the parallelism cannot span two or more function calls. On the contrary, in order to suit the event-driven programming model and for all event handlers to co-use a parallel region, the task executor model is used. A virtual target is essentially a thread pool executor, or an event dispatching thread, and its lifecycle lasts throughout the program. Conceptually, a virtual target represents a type of execution environment defining its thread affiliation (to ensure operations not thread-safe are only executed by a specified thread), and scale (confines the number of threads of a thread pool). This design enables programmers to flexibly submit different code snippets to different execution environments. This section describes the additional OpenMP APIs supported by Pyjama, which are used for managing virtual targets at the runtime.

Every virtual target used in the directive requires either a registration or a creation (Table 4). For example, in a Java Swing GUI application, the master thread is the Event

<table>
<thead>
<tr>
<th>Name</th>
<th>virtual_target_register()</th>
<th>virtual_target_create()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>tname: String</td>
<td>tname: String, n: Integer</td>
</tr>
<tr>
<td>Description</td>
<td>The thread which invokes this function will be registered as a virtual target named tname.</td>
<td>Creating a worker virtual target with maximum of n threads, and its name is tname.</td>
</tr>
</tbody>
</table>

Table 4: Runtime functions to create virtual targets in Pyjama.
Dispatching Thread. In order to notify the compiler regarding the master thread as a virtual target \texttt{edt}, a registering function \texttt{virtual\_target\_register(“edt”)} should be executed at the initializing stage (e.g. the constructor of the graphic interface). Similarly, creating a new virtual target can be achieved by a creation function. For instance, invoking the function \texttt{virtual\_target\_create(“worker”, 5)}, will create a new virtual target called \texttt{worker} which has a maximum thread number of 5.

__Registering dispatching mechanisms__

Registering an EDT as a virtual target requires specifying the task dispatching mechanism to the Pyjama runtime, otherwise the runtime cannot post runnable tasks to the EDT targets. However, the dispatching mechanism depends on the GUI framework. In the current experimental version of Pyjama, three types of event-driven GUI framework are already supported: Java Swing [95], Android [96] and JavaFX [97]. For other frameworks, programmers are required to specify the task dispatching interface that the framework provides.

__The distinctions between omp task and omp target virtual__

The virtual target concept allows programmers to easily change the thread context, and submit the code blocks to a different thread pool, without knowing any underlying implementation details. The salient advantage of using virtual targets is its compatibility with an event-driven framework. For most event-driven frameworks, only the interfaces of event handlers are exposed to the programmers, and programmers cannot directly modify the dispatching mechanism. Under this circumstance, using OpenMP task directives shows its disadvantage because a task is only active when it is within a parallel region, but the programmers cannot use the parallel directive to parallelize the dispatching framework [98].

Listing 11 and Listing 12 show two simple examples of using this approach. The example of using OpenMP tasks forces the code change onto the event loop, then asynchronization of the event handlers becomes possible. In contrast, for virtual targets, programmers can directly use the target virtual directive inside event handlers to offload computations away from the event handling thread. Another distinction of these two concepts is that with OpenMP tasks, the master thread is a part of the thread group. In comparison, with virtual targets, the master thread and worker threads are explicitly
Listing 11: An example of using OpenMP tasks.

```c
void server()
{
    #pragma omp parallel
    {
        #pragma omp single
        while(1)
        {
            #pragma omp task
            event_handler1();
            #pragma omp task
            event_handler2();
        }
    }
}
```

<table>
<thead>
<tr>
<th>Distinction</th>
<th>OpenMP Tasking</th>
<th>Virtual Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>Task parallelization</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Scenario used</td>
<td>Task decomposition and parallelization</td>
<td>Event-driven offloading and context switching</td>
</tr>
<tr>
<td>Effective region</td>
<td>Only in parallel region</td>
<td>Everywhere</td>
</tr>
<tr>
<td>Dependency handling</td>
<td>Data dependency</td>
<td>Control flow dependency</td>
</tr>
<tr>
<td>Task pool number</td>
<td>Single</td>
<td>Multiple</td>
</tr>
</tbody>
</table>

Table 5: Comparison between OpenMP tasking and target virtual.

distinguished. If the current thread is not a member thread of the virtual target’s thread pool, the target code block will not be executed by the current thread.

Another salient difference between virtual target and OpenMP tasks is their runtime thread pool controls (Figure 26). With OpenMP tasks, only one thread pool is managed to which all tasks are enqueued to a single queue. In contrast, more than one task queue can be managed in the virtual target programming model. Programmers can submit task code blocks to different pools according to the properties (e.g. execution times). This is effective in boosting performance, which will be demonstrated in Section 5.3.

A summary of the differences between OpenMP tasks and virtual target is listed in Table 5.
Listing 12: An example of using virtual targets.

```c
void event_handler1()
{
    //#omp target virtual(worker) await
    compute_half1();
    //#omp target virtual(edt) nowait
    notify("Task half finished");
    //#omp target virtual(worker) await
    compute_half2();
    //#omp target virtual(edt) nowait
    notify("Task finished");
}
```

Figure 26: Demonstration of OpenMP tasking (a) and virtual target (b).
IMPLEMENTATION OVERVIEW

This section discusses the implementation of the proposed programming model in Pyjama, which is an OpenMP-like implementation for Java. Pyjama mainly constitutes two parts. First is the source-to-source compiler which supporting traditional OpenMP directives and the extended directives this work proposed, transforming the sequential Java source code into parallel code. Second is the runtime system, which provides the underlining thread-pool creation, management and task scheduling, as well as all the OpenMP runtime functions.

Compilation

This section provides an overview of how an OpenMP target block or an async method is converted to the destination code.

Auxiliary class generation

An auxiliary class is an inner class of the current compilation unit, which contains all the running information of a target block or an async method. In general, each target virtual code block is refactored into an inner class, and this inner class inherits an abstract class called TargetTask. The abstract interface call() is implemented to include the user code. Meanwhile, the class contains several data fields which store the information and track the execution status of this target block. Figure 27 illustrates all the noticeable fields. Meanwhile, all the variables which are used in the target block are also required as field variables in the auxiliary class, and they should be passed in and initialized by the auxiliary class constructor.

Generating code with states

If a target block or a method contains any await target blocks, or await constructs, the generated auxiliary class may contain states. The interface call() will be implemented with states, by which the control flow of call() can be resumed to different positions according to the state number. This makes the target block or async function flexible enough to suspend and resume during its execution at appropriate continuation points.

```c
//omp async
int asyncCall()
{
```
public abstract class TargetTask<T> implements Callable<T>{
    //which virtual target should invoke this task.
    private VirtualTarget caller;
    //the callback function should be triggered when this task finishes.
    private CallbackInfo callWhenFinish;
    //the flag indicating if this task is finished.
    private volatile boolean isFinished;
    //the return value, only available for async functions.
    private T result;
    //the current state of the execution.
    private int state;
    //the interface the subclass should implement.
    public abstract T call() throws Exception;
}

class CallbackInfo {
    //the continuation call.
    TargetTask<?> callback;
    //who calls this continuation.
    VirtualTarget caller;
}

Figure 27: An overview of the TargetTask Class.
int result;
//#omp await (int foo(int a))
{
    result += foo(1) + foo(2);
}
//#omp target virtual(edt)
{
    Panel.update(result);
}
return result + bar(1);

For each await invocation, an instance of its paired auxiliary class is initialized with arguments. The entire process is separated by states. The state of the control flow cannot process until the current awaiting call is finished. During this waiting, the control flow returns back to its caller. By using an on-completion-handler, the control flow resumes to the appropriate continuation point according to its current state. Finally, at the end of the process, the result of the async method is settled by using the setResult() method:

```java
public void call() {
    switch(OMP_state) {
    case 1:
        OMP_AwaitFunctionResult_foo_0 = new _OMP_StateMachine_foo(1);
        OMP_AwaitFunctionResult_foo_0.setOnCompleteCall(this,
            PjRuntime.getVirtualTargetOfCurrentThread());
        PjRuntime.runTaskDirectly(OMP_AwaitFunctionResult_foo_0);
        this.OMP_state++;
        return null;
    case 2:
        OMP_AwaitFunctionResult_foo_1 = new _OMP_StateMachine_foo(2);
        OMP_AwaitFunctionResult_foo_1.setOnCompleteCall(this,
            PjRuntime.getVirtualTargetOfCurrentThread());
        PjRuntime.runTaskDirectly(OMP_AwaitFunctionResult_foo_1);
        this.OMP_state++;
        return null;
    case 3:
        result += OMP_AwaitFunctionResult_foo_0.getResult() +
            OMP_AwaitFunctionResult_foo_1.getResult();
        OMP_TargetTaskRegion_0 = new _OMP_TargetTaskRegion_0();
    ```
**TARGET BLOCK INVOCATION**

In the generated code, the invocation of every target block or async method is converted to the invocation of its paired auxiliary class. First, an instance of its auxiliary class is initialized, with proper arguments. Second, the block is scheduled by the runtime routine according to its `asynchronous-property-clause`. For example, consider the following code snippet:

```java
Label.setText("Start Processing Task!");
//#omp target virtual(worker) await
{
    compute_half1(); // S1
    //#omp target virtual(edt) nowait
    {
        Label.setText("Task half finished"); // S2
    }
    compute_half2(); // S3
}
Label.setText("Task finished"); // S4
```

The `call()` interface is implemented by code inside the target code block, in favor of generating the auxiliary class which extends `TargetTask`. The data context and variables referenced by the user code are stored into this generated class (for simplicity, the demo code omits the field variables). The target region instance is then submitted...
to the Pyjama runtime, which is responsible for dispatching the target code block to the appropriate virtual target.

```java
class TargetRegion_0() extends TargetTask {
    public void call() {
        compute_half1(); // S1
        TargetRegion _omp_tr_1 = new TargetRegion_1();
        PjRuntime.invokeTargetBlock("edt", _omp_tr_1, Async.nowait); // S2
        compute_half2(); // S3
    }
}
Label.setText("Start Processing Task!"); // S4
TargetRegion _omp_tr_0 = new TargetRegion_0();
OMP_TargetTaskRegion_0.setOnCompleteCall(this, "worker");
PjRuntime.invokeTargetBlock("worker", _omp_tr_0, Async.await);
Label.setText("Task finished");
```

**Runtime**

The runtime support includes the runtime functions and the underlining target task dispatching mechanism.

**Target block scheduling**

During execution of the program, target blocks are dynamically dispatched by the Pyjama runtime. The logic of invoking a target block is presented in Algorithm 4.1. The runtime routine first checks if the submitting thread is already a member of the virtual target executor’s thread group (line 7). If yes, it means the target block is already in the context of the virtual target execution environment, so it is executed synchronously by the current thread (line 8). If the asynchronous-property-clause is nowait or name_as (line 11), the main thread exits the procedure (line 12) to directly execute the statements following the target block. If the asynchronous-property-clause is await, the caller suspends its execution and once the target block is finished, the caller continues its execution from where it was suspended, which will be executed by the same thread or executor (lines 15-17). Otherwise, the thread waits for the target block to finish (line 18).
Algorithm 4.1 Target block code execution.

1: $T$: current thread
2: $C$: current caller function
3: $E$: target executor
4: $B$: target block
5: $a$: asynchronous property
6: procedure `invokeTargetBlock($T, C, E, B, a$)`
7:     if $T \in E$ then
8:         $B$.exec() \quad \triangleright \text{execute } B \text{ synchronously by } T
9:     else $E$.post($B$) \quad \triangleright \text{post } B \text{ to } E \text{ asynchronously}
10:    end if
11:    if $a$ is `nowait` or `name_as` then
12:        return \quad \triangleright \text{directly return to caller}
13:    end if
14:    if $a$ is `await` then
15:        $C$.suspend()
16:        $B$.setOnCompleteCall($C, T$) \quad \triangleright \text{continuation binding}
17:        return
18:    else $T$.wait() \quad \triangleright \text{default option}
19:    end if
20: end procedure
Post to different virtual targets

Algorithm 4.2 Post target block to different types of virtual target.
1: $E$: target executor
2: $B$: target block
3: procedure $\text{post}(E,B)$
4: \hspace{1em} if $E$ is registeredEDT then
5: \hspace{2em} switch $E$ do
6: \hspace{3em} case Swing
7: \hspace{4em} SwingUtilities.invokeLater((Runnable)$B$);
8: \hspace{3em} case Android
9: \hspace{4em} Handler uiHandler = new Handler(Looper.getMainLooper());
10: \hspace{4em} uiHandler.post((Runnable)$B$);
11: \hspace{3em} case JavaFX
12: \hspace{4em} Platform.runLater((Runnable)$B$);
13: \hspace{2em} else $E$.post($B$) \hspace{4em} $\triangleright$ post $B$ to a non-EDT target
14: \hspace{2em} end if
15: end procedure

Using Pyjama runtime requires a mechanism to dispatch events and tasks to the EDT. The approach of posting tasks to the EDT is decided by the particular GUI framework and platform. Figure 4.2 reveals how the current Pyjama version supports three different types of GUI frameworks. The Pyjama compiler generates the appropriate target code according to the platform the programmer specifies.

Exception handling support

In order to support the asynchronous exception handling, the user code is surrounded by a try-catch block in the implementation of the $\text{call()}$ function. Any exception escaping from the target virtual block is stored by the runtime, and then the target block is marked as completed. After any appropriate synchronization point is reached, the $\text{getResult()}$ function is invoked. Before returning the result, the runtime checks for any stored exception. If yes, the exception is re-thrown at this point, and then the encountering thread is able to handle it.

```java
public void call()
{
    try{
```
EVALUATION

This section provides the evaluations of the proposed approach for event-driven programming. Three case studies are presented.

Java GUI event handling

Modern real-world applications/apps usually require a high computational ability without losing any responsiveness. For example, consider a mobile visual-realism application constantly capturing images from a camera and then applying the image rendering or processing (e.g. augmented reality) for the user. In order to achieve a smooth user experience, the processing of each frame should be as short as possible, especially when many images are captured in a short period. Here, scenarios are simulated in which a GUI application is under different loads of event handling, and the benchmarks measure the ability of handling events by different approaches.

The first evaluation compares the different methods for offloading time-consuming work to the background, while maintaining a responsive GUI. Since the benchmarks are performed under the Java Swing GUI framework, three different approaches are compared: SwingWorker, ExecutorService (using SwingUtilities when necessary) and Pyjama. Each benchmark adopts a computational kernel selected from the Java Grande Benchmark suite [99] (since the kernel can be parallelized by using traditional OpenMP directives), to simulate the time-consuming computational work within event handlers.
Figure 28: Average event response time, as a proportion of the sequential version, using different offloading approaches and computational kernels (lower is better).

Selected were Crypt, RayTracer, MonteCarlo and Series. There are GUI updates before and after the kernel execution. As discussed before, those GUI related operations are required to be executed in the EDT. As the application utilizes a GUI component, the benchmarks are performed on a typical desktop machine (in this case an i5-3570 quad-core Intel processor, with 8M cache, up to 3.90 GHz clock rate). Oracle’s Java 1.8.0_66 VM is used throughout the benchmarks.

The benchmarks are categorized by the kernels. For each benchmark, the event is bound with an execution of its kernel. Every benchmark is run 10 rounds with different request loads, ranging from 10 requests/sec to 100 requests/sec. The response time shows the time flow from the event firing to the finish of its event handling. The average response time of all events shows a general efficiency of processing of event handling.

To show how different approaches decrease the average response time, compared to the sequential version, different offloading approaches are presented. The results also show the synchronous parallel version (in default using 3 worker threads), in which only the computational kernels are parallelized and the EDT still does part of the computing job when handling the events. Therefore, the EDT in the synchronous parallel approach is actually unresponsive for a longer time compared to other approaches. The underlying implementation of SwingWorker maintains a default 10-thread-max thread pool. Figure 28 depicts the results, showing the average response times in proportion to sequential versions. The results show that Pyjama has a comparable (or in some cases better) event response time compared to the other manual approaches, especially when three worker threads execute the kernels in the background. It is also interesting to observe that
the execution of kernels in parallel (but synchronously) is inferior to an asynchronous execution with the same number of threads when comparing the response times.

In a GUI application, if responding to an event trigger exceeds 5 seconds, the application is deemed unusable [87]. Using this rule of thumb, Figure 29 counts the number of event responses that complete within 5 seconds, depending on different event request loads. Event requests are kept consistent for each sequential version, since it reaches the maximum handling ability of single-threaded sequential versions. SwingWorker shows inconsistent performance as the request load differs, which may be attributed to its underlying scheduling policy of its thread pool tasks. The ExecutorService shows a performance degradation when more events happen in the same time unit. It may be attributed to the accumulated overhead by task submissions of the underlying implementation of ExecutorService. In contrast, Pyjama’s virtual target offloading keeps a consistent and high response rate. This shows that the implementation of Pyjama’s runtime is more suitable for offloading more tasks under the scenario of heavy workloads of event handling.

**Responsiveness of the EDT**

To study the responsiveness of the EDT, an interactive application was developed in Java called ParaImage [100], which is a desktop application for image search and manipulation.

For a very good user experience, the programmer wants to make the GUI interface very responsive by offloading long running tasks to background threads. Many approaches are available, such as SwingWorker or SwingUtilities in standard Java. However, these approaches all require substantial code refactoring and advanced knowledge [80]. When doing this with OpenMP-like directives, the parallelization is virtually free and the code logic remains sequential. To illustrate this, Listing 13 presents an event handling function that when a search button is clicked, the application starts searching the images from the Internet according to the keyword. The implementation shows the sequential version augmented with OpenMP directives (`#omp ...`). If the code is compiled through the Pyjama compiler, the OpenMP comments will be triggered and the executable bytecode will run in parallel, otherwise the directives are treated as comments and ignored.
Figure 29: 5-second response count under different request work loads (higher is better).
Listing 13: The search listener function implementation.

```java
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == searchButton) {
        String keyword = textField.getText();
        int resPP = (Integer)spnResultsPerPage.getValue();
        setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
        //#omp target virtual(worker) name_as(search)
        {  
            PhotoList list = PhotoInterface.search(keyword, resPP, currentOffset);
            for (int i=0; i<list.size(); i++) {
                PhotoInfo photo = list.get(i);
                PhotoWithImage image = new PhotoWithImage(photo);
                //#omp target virtual(edt) nowait
                {  
                    panel.progressBar.setValue(i / list.size() * 100);
                    panel.addToDisplay(image);
                    panel.updateUI();
                }
            }
            //#omp target virtual(edt) nowait
            {  
                progressBar.setValue(0);
                thumbnailsPanel.updateUI();
                setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
            }
        }
    }
}
```
In order to profile the latency of the graphical user interface, an instrumentation tool LagHunter [85] is used to trace the call stack of the EDT in the runtime. The output of LagHunter is analyzed post-mortem with LagAlyzer [101], a latency profile analysis and visualization tool that is used to characterize the noticeable lags from the tracing data.

To illustrate this, a searching scenario is taken as an example. ParaImage provides a search interface where users can retrieve images from the Internet by inputting key-words and the thumbnails of the results appear in the display panel. When the feature is implemented in a sequential, synchronous way, the users will experience an unresponsive interface until all the results are retrieved. During this time, none of the buttons or menus will respond. A quick asynchronization can be applied by adding proper virtual target directives into the sequential code, and partial results in the form of thumbnails will be displayed as soon as they become available. Other GUI elements such as the cancel button, remain enabled. Unlike the sequential, synchronous version, users are now able to cancel the remainder of the retrieval midway during the search.

LagAlyzer visualizes the entire lifetime activities of the EDT. The x-axis represents the time elapse from the application starts. The y-axis shows the call stack of the EDT at the specific time. Figure 30 shows the profiling visualization of the EDT in both the sequential and asynchronous versions. The visualization traces the method call from the EDT and methods are categorized in different colors according to their properties. Table 6 explains the details. In the sequential implementation, the dispatching clearly
Table 6: Interval types and their represent colors.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch</td>
<td>EDT is busy on dispatching events</td>
<td>Red</td>
</tr>
<tr>
<td>Modal</td>
<td>EDT is handling a modal window</td>
<td>Orange</td>
</tr>
<tr>
<td>Listener</td>
<td>EDT is handling an event listener synchronously</td>
<td>Pink</td>
</tr>
<tr>
<td>Asynchronous call</td>
<td>Handling of an event posted asynchronously from other thread</td>
<td>Green</td>
</tr>
<tr>
<td>Paint</td>
<td>Graphics rendering operation</td>
<td>Blue</td>
</tr>
</tbody>
</table>

covers large parts of the lifetime of the EDT. A click of the search button leads to a synchronous processing and the EDT remains unresponsive until the event handling is completed. It can be seen from Figure 30a the most of the lifespan of the EDT is occupied by long time function calls. In contrast, in the asynchronous version, when the search button is triggered, the search is executed in the background because the execution is offloaded to a virtual target. During this period of time, the EDT remains idle which means the application remains responsive. Only necessary GUI updates are executed in the EDT (which only the EDT is allowed to do) by asynchronously posting from the virtual targets to the EDT virtual target. Therefore, in Figure 30a, only a series of minor function calls are executed by the EDT and they are all UI related operations. In this search scenario, after applying the OpenMP directives, the EDT idle rate increases from 74.5% to 91.0% compared to the sequential version (higher is better).

According to the Human-Computer Interaction study on the quantification of the user perceptual time, the human perceptible response time to the GUI is above 100ms [102]. By applying this threshold as the criterion, a Cumulative Latency Distribution [103] is given in Figure 31 for the analyzed ParaImage. The x-axis shows the latency in milliseconds, and the y-axis indicates how many user requests are taking longer than the given x ms in this application. An ideal curve would be deep L-shaped. From the figure, it can be discovered that if the application is not made asynchronous, around 70 out of 170 user requests are longer than 100ms, which causes a noticeable bad user experience. After applying asynchronization using the virtual target OpenMP directives, the result curve shows a nearly ideal shape, turning the application into a rich interactive style.
Web service event handler

The purpose of this benchmark is to evaluate the scalability of Pyjama’s virtual targets runtime for a different type of event. The experiment implements an HTTP service that provides data encryption to web users. Every time a user sends input data with an HTTP request, the server performs a calculation and returns the result via the HTTP response. The encryption computation can be parallelized by adopting traditional OpenMP directives. The web server is implemented using two approaches. The first uses Pyjama’s virtual target to offload the time-consuming computations to worker threads. The second uses Jetty’s [104] thread-pool framework, which adopts a thread-per-request policy but reuses a fixed number of threads from a thread pool. This experiment is run on a 16-core Intel Xeon 2.4GHz SMP machine with 64 GB memory, and Java 1.8.0_66 HotSpot 64-Bit Server VM.
The load benchmark is set up with 100 virtual users, with each user sending a constant number of requests. The throughput measures the application’s ability to process requests. Figure 32 describes that both Jetty and Pyjama have good scaling performance as the number of concurrency worker threads increases. When the parallelization of each event (using `//#omp parallel`) is used in combination with either Jetty or Pyjama, it initially results in a dramatically better throughput. Yet, as the number of concurrency worker threads is increased, the throughput levels off at just under 50 responses/sec. The non-parallelized versions achieve better throughput when the number of concurrency workers gets above 13. This result is reasonable, because every parallelization computation spawns its own set of worker threads. With the increased amount of computation requests, the total number of threads in the system soars to a high value and it leads to a great overhead of thread scheduling.

### Android application

This section presents a case study of using Pyjama on the Android platform. The purpose of this case study is to evaluate the effectiveness of using Pyjama to boost the performance and responsiveness of a mobile application.

The scenario of this experiment is using a demo Android app to process the images collected from the camera device on the fly. When the app starts, the program constantly picks one frame from the camera frame buffer and applies a gray scale image processing using Catalano Framework [105], then the app updates the result of the processing to the user interface. This application was evaluated on a Nexus 4 with 1.5 GHz quad-core, running Android 4.4.4 OS. For a single round of the image processing, in which the frame size is 460800 bytes, it takes around 200ms to convert camera compatible format NV21 to image processing library compatible format Bitmap. Then the gray scale processing takes approximately 250ms and the UI update takes about 20ms (Figure 33).
Supporting Asynchronization in OpenMP for Event-Driven Programming

The event handler is implemented in three different approaches, and for each the LoC (Lines of Code) and FPPM (Frames Processed Per Minute) are compared in Table 7. The first approach is the single-threaded implementation, in which case the EDT is responsible for all the background computations and UI updates. This leads to low performance and bad responsiveness of the UI, since the UI is frozen during the background computation. The second approach uses Android’s AsyncWorker class to offload the computation asynchronously to a background thread. While this gains better performance in terms of FPPM, it requires code refactoring. The final approach, using Pyjama, gains the competitive performance improvement but without the refactoring effort required in AsyncWorker.

Since Pyjama is directive-based, it promotes an incremental programming approach enabling programmers to retain the sequential code. Therefore, it avoids code restructuring, the variable scope and programming context also do not change. In summary, Pyjama is effectively automatically generating asynchronous code native to the application. This means programmers witness the equivalent performance boosts they can expect from a manual refactoring, but with minimal programming effort over the original single-threaded implementation.

Related Work

Asynchronization

Asynchronous programming is traditionally used in single-threaded applications to achieve cooperative multitasking [106]. Unlike parallel programming that creates multiple threads, this programming model employs a single background thread. As such, the purpose of introducing asynchronization is not to make the program run faster. Instead, it is used when an event handling thread needs to wait for time-consuming computations or I/O. In this manner, the thread can still progress since the control flow is switched to another task.

Libraries. Many languages provide built-in or extended library interfaces to support asynchronous programming. For example, C++11 provides std::async, while Java provides the Future interface [107] building asynchronous computations. Java NIO libraries [108] provide non-blocking and asynchronous I/O operations. Microsoft .NET provides three types of asynchronous programming patterns [109]: (1) Asynchronous
<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of Code</th>
<th>Refactoring</th>
<th>Frames Processed Per Minute</th>
<th>Responsive UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDT (single thread)</td>
<td>17</td>
<td>-</td>
<td>78</td>
<td>no</td>
</tr>
<tr>
<td>AsyncWorker</td>
<td>29</td>
<td>yes</td>
<td>103</td>
<td>yes</td>
</tr>
<tr>
<td>Pyjama</td>
<td>19</td>
<td>no</td>
<td>106</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 7: A comparison of three approaches, with regard to programming effort and performance.
SUPPORTING ASYNCHRONIZATION IN OPENMP FOR EVENT-DRIVEN PROGRAMMING

Programming Model (APM); (2) Event-based Asynchronous Pattern (EAP); (3) Task-based Asynchronous Pattern (TAP).

Frameworks. The implementation of an asynchronous task usually applies an event-driven programming pattern, of which the continuation of the task is transformed as a callback function which will be triggered when asynchronous operations finish. This idea has been adapted to many different languages and frameworks, especially for the sake of high-performance network server developing. For example, libevent [110] is an asynchronous event-based network application framework written in C, adopting the proactor pattern [111], which is an object behavioral pattern of the combination of I/O multiplexing [112] and asynchronous event dispatching. Similarly, other frameworks written in other languages (e.g. [113, 114, 115]) have become increasingly popular in recent years.

Language support. Unlike libraries, language-level support for asynchronization tends to require less code restructuring. Fischer et al. [116] proposed TaskJava, a backward-compatible extension to Java. By introducing new keywords (i.e. spawn, async, wait), TaskJava expresses the complicated asynchronous logic control flow using intuitive sequential programming style. Similarly, the .NET framework also introduces paired async/await keywords [117]. New language designs also tend to support asynchronization. For example, P [118] is a domain-specific language for the modeling of state machines, and all machines communicate via asynchronous events. Eve [119] is another parallel event-oriented language for the development of high-performance I/O applications. Other language-level concepts such as the actor model [120, 121] and co-routines [122] provide variations to asynchronization. S [123], a domain-specific language targeting the server-side scripting of high-performance RESTful Web services, is based on the JavaScript asynchronous execution model. TigerQuoll [124] is a parallel event-based runtime for programming in JavaScript, which supports an innovative transaction model to speed up high-contention workloads.

Task-based Parallelism

The task-based parallelization model is usually implemented to overcome the performance issues of the threading model. A fixed thread pool substitutes preemptive thread-creation when a computational task is needed. The thread pool technique encapsulates the underlying threading and scheduling [125] and provides interfaces for
task submissions. Some languages support tasks at a language level, such as Cilk [26] and JClk [126]. While OpenMP provides the task directive [32], the lifetime of a task is confined inside a parallel region. In addition to the actual parallelization, handling task dependencies and code restructuring is another challenge faced. Parallel Task [76], as a language extension of Java, supports task creation and dependency handling. OoOJava [127] and DOJ [128] both introduce the task keyword to achieve out-of-order execution of the code blocks, with the support of automatic dependency analysis between tasks. TWEJava [129] is another extended version of Java that supports flexible Java concurrency and tasking programming while delivering very strong safety properties including task isolation, data race freedom, atomicity, and optionally determinism.

**SUMMARY**

This chapter proposed a hybrid model for the combination of asynchronization and parallelization, as an extension of OpenMP. The idea is implemented in Pyjama, an OpenMP compiler and runtime support for Java. The model facilitates the development of event-driven programs, especially for GUI applications, to achieve better responsiveness and event handling acceleration. Strictly following the philosophy of OpenMP, the semantic design of this model does not interfere with the original sequential programming logic. With the help of a supporting compiler, the additional directives generate event handling code to execute asynchronously and offload computations away from the event dispatching thread. Evaluations show that single-threaded event dispatching can be quickly upgraded to a higher performing multi-threaded event dispatching, by reducing event handling response time. Performance achieved by the proposed directive based approach is equal and often superior to manual implementations. The current virtual target is not mature enough to handle I/O-bound operations. An ideal I/O virtual target implementation requires a few number of threads that are able to handle a great amount of I/O operations by multiplexing. This is a future optimization point of Pyjama implementation that supports high-performance I/O operations.
A previously proposed extension has made it possible for OpenMP to speedup event-driven programs. There, a virtual target model extension is used to incrementally introduce asynchronous execution into an OpenMP program. At the same time, it allows a mixture or nested parallelism with asynchronous processing. This new possibility raises the question of how to use available processors/threads, for parallelism or for asynchronous execution. To investigate the best combination of asynchronization and parallelization, a performance model for measuring parallel event-driven systems is proposed. Based on queue theory, the theoretical analysis discovers some interesting facts in an event-driven system. Experiments are conducted to study the best practice of improving event-driven programs, and how to balance parallelism and asynchronous execution. By comparing it with the OpenMP tasking model, the evaluations demonstrate the effectiveness and flexibility of the virtual target model, which is able to achieve significantly better parallel event-driven performance.

The contribution of this chapter is twofold:

- A performance model for event-driven parallelization is proposed. In this model, factors that influence the event-driven performance are profiled. This model gives a theoretical reference for the benchmarks.

- An experimental evaluation demonstrates the effectiveness of using the virtual target concept for an event-driven framework. It offers more usability and flexibility compared to the traditional OpenMP tasking concept. In particular, its design enables categorizing different tasks, to be submitted to different thread pools.
according to their sizes or run time, which thereby minimizes the mean event handling flow time.

The contents of this chapter are based on the published paper in 2017 13th International Workshop on OpenMP [130], and a recently submitted journal paper.

The structure of this chapter is as follows: Section 5.1 provides the introduction. Section 5.2 describes the theoretical background of parallelizing event-driven programs and a mathematical model is presented to quantify the performance. Section 5.3 presents the experiments performed and the results obtained, and demonstrates the effectiveness of using the proposed programming model and performance model. Section 5.4 summarizes.

INTRODUCTION

A wide range of modern applications are developed based on the event-driven model, ranging from mobile apps, desktop applications and web services. In general, these types of programs have an interactive nature, which means their execution is not predetermined, but rather depends on the events or requests that arise during runtime.

In an event-driven framework, a particular thread is solely responsible for driving the event-loop, dispatching the events and calling the event-related handlers: the Event Dispatching Thread (EDT). By default, for a naive implementation without any multithreading applied, the EDT executes all the queued event handlers in a sequential manner. A problem emerges if the event handling functions are time-consuming, or registered events suddenly burst within a short period of time. With such execution burdens, the EDT is unable to handle more upcoming events, thus leading to an unresponsive application.

Integrating parallelization and asynchronization for event-driven programming with such a construct raises interesting questions. There is no widely used performance model relating to the parallelization of event-driven programs. As a consequence, it is not clear how code parallelization influences the performance of an event-driven application. Figure 34 shows a situation that in an event-driven system, some processors are distributed as asynchronous workers to offload computations for the event queue whereas some other processors are distributed to parallelize some event handlers. As a consequence, an interesting question arises: if the event handlers have the potential to be parallelized, is it worthwhile to parallelize the handler functions and how many threads
should be used in the parallelism? Since computational resources are usually limited, the assignment of processors can strongly influence the performance. This chapter attempts to address these questions with a performance model and an experimental evaluation.

By adapting the queue theory model, a performance model for parallel event-driven programming is presented. Under a theoretical analysis, some interesting facts about the parallelization of event-driven programs are discovered. These facts can help programmers make decisions on choosing the proper size of asynchronous workers to offload different event handlers to the background. At the same time, if the event handler itself can be parallelized, the performance model is also useful to decide the most efficient parallelization scale.

**PERFORMANCE MEASUREMENT OF EVENT HANDLING EXECUTION**

In an event-driven system, simply measuring speedup and execution time to evaluate the performance of parallelism is not appropriate because events become available or are released at unknown times, which can be modeled stochastically. Hence it needs some other measure to handle events in a system. For this reason, this section establishes a performance model based on queue theory.
Flow time of event handling

Going by the standards of GUI frameworks, the proposed model assumes that only the EDT is responsible for dispatching the events, and an event request queue is maintained by the EDT. The flow time $t_F$ measures the time span from the triggering of the event to the finish of its related event handling. The notation $t_R$ measures the residual service time of the current event handler that is under processing. The queuing time $t_Q$ indicates the handling function cannot process until all previous queued handling functions are complete. Afterwards, the service time $t_S$ is conducted for the processing of this event handling.

Given an event binding $e \rightarrow F$ in a system. When the event $e$ happens, its event handling function $F$ should be triggered. At the time the event $e$ is triggered, there are potentially unprocessed event handlers in the event queue, and the set $F$ represents all queued event handlers at the triggering point of $e$. The flow time $t_F$ of the event handling is the sum of three parts: the residual time $t_R$ of the event handler which is under processing at the moment, the event $e$’s queuing time, and its handling function $F$’s execution time (service time) $t_S$:

$$t_F = t_R + t_Q + t_S = t_R + \sum_{f \in F} t_S(f) + t_S$$

If no concurrency (asynchronization) is employed within the event handling function, the execution is solely executed by the EDT. As a consequence, the EDT cannot respond to other events or requests during the time period $t_F$. If $t_F$ is long enough, users will experience a degraded usability.

Processing events in a multi-threaded environment

For an application in a multi-threaded system, two approaches to reduce the flow time $t_F$ of each event handling are possible. The first approach is enabling the system to have multiple asynchronous workers to process the queued requests, then $t_Q$ is reduced. The second approach is to parallelize the event handlers, which reduces their execution time in comparison to sequential execution, decreasing the service time $t_S$ and in turn the $t_Q$. Denote $N_a$ as the asynchronization scale, and $N_p$ as the parallelization scale. Ideally, assume the asynchronous workers do not suffer from any performance degradation and increasing the number of asynchronous works always gets $N_a$ speedup; in other
words, the execution load of all event handlers to be executed concurrently is ideally balanced across the $N_a$ workers. On the other hand, due to the typical nature of the handling functions, it is often not that ideal speedup is achieved when parallelizing the event handlers. Therefore define parallelization efficiency function as $\epsilon(N_p) = \eta N_p$, in which the handling function gains $\epsilon(N_p)$ speedup when using $N_p$ threads to parallelize this event handler. $\eta$ is the parallelization efficiency factor.

$$t_F = t_R \frac{\epsilon'(N_p)}{\epsilon'(N_p)} + \sum_{f \in F} \frac{t_Q(f)}{\epsilon_f(N_p)} N_a^{-1} + \frac{t_S(F)}{\epsilon(N_p)}$$

Ideally, the handler execution can be totally offloaded to the background thread, but sometimes it is necessary for the EDT to handle some UI updates exclusively. For example, in a GUI application, all GUI update manipulations must be executed by the EDT. Under this scenario, the flow time in an event handler can be decomposed into two parts: the execution by the EDT and the execution by the asynchronous workers:

$$t'_F = t_{ui} | t_{edt} + t_F | t_{worker}$$

Offloading effectively decreases the handling time of the EDT, because the EDT is only responsible for handling UI related operations. Therefore, the EDT has more idle time for the subsequent event handling, and the responsiveness of the application is improved. Under this circumstance, only the operations that executed on background threads have the potential to be parallelized.

**Modeling of the parallel event-driven system**

As mentioned, there is no widely used performance model relating to the parallelization of event-driven programs. As a consequence, it is not clear how parallelization influences the performance of event-driven executions. An interesting question arises if the event handlers have the potential to be parallelized. Every available thread (processor) in the system can be used to parallelize the event handlers, or alternatively it can be used as an asynchronous worker to where the EDT offloads computations. The partition of the available threads (processors) can significantly influence the performance of the event-driven system.

To model the parallel event-driven system, the model employs the Kendall Notation that is used to describe queuing systems [131]. An event-driven system can be described as such a queuing system. Kendall proposed describing queuing models using three
factors written $A/S/c$, where $A$ denotes the time between arrivals to the queue, $S$ is the size of jobs and $c$ is the number of servers at the node. As a default, assume that this model has unlimited capacity of the queue, and the queuing principle is First In First Out (FIFO).

- $A$: Arrival process
  - $M$: The arrival process is governed by a Poisson Distribution
- $S$: Service time distribution
  - $M$: The service time is exponentially distributed
  - $D$: The service time is deterministic, which is a constant value
- $c$: Number of servers

Queue model with parallelism

Now extend the $A/S/c$ model to integrate parallel execution of the jobs (event handlers). Define $N_a$ as the asynchronization scale of the multi-core machine (corresponding to $c$), and $N_p$ as the parallelization scale of a multi-core machine. The maximum threads (processors) in this system is $N_a N_p$. Then the $A/S/c$ model is extended as $A/S/N_a/N_p$.

Because for most of the event-driven systems, every event comes independently with a specific arrival rate, which can be described as Poisson Process, suppose the arrivals of the event requests are governed by a Poisson Distribution \cite{132}, and the sequential handling times for the handlers are exponentially distributed. In this parallel queue system, there are $N_a$ multiple asynchronous workers that can process the requests at the same time, and there are $N_p$ parallel threads in a paralleled handling function. This model is described as $M/M/N_a/N_p$.

In order to better analyze the performance of this model, the factors related to this model are listed as below:

- $\lambda$ is the mean arrival rate of the requests/events.
- $\mu$ is the mean sequential service rate; if the mean sequential service time of the handlers is $T_{seq}$, then $\mu = \frac{1}{T_{seq}}$.
- $L_q$ is the mean waiting queue length.
- $L$ is the mean queue length, including the handlers in service.
• $W_q$ is the mean waiting time in the queue.

• $W$ is the mean flow time spent at the queue both of waiting and being serviced.

• $N_a$ is the asynchronization scale, which is the number of asynchronous workers in the queue system.

• $N_p$ is the parallelization scale, which is the number of parallel threads in event handler’s parallel region.

**Utilization.** Define the service utility as $\rho$, which presents the utilization of processors. Then:

$$\rho = \frac{\lambda}{\mu N_a c(N_p)}$$

(1)

The utilization measures the occupation of the processors. If this value is too low, it means the incoming tasks do not create a high usage of the computation resources. A good use of a parallel system is keeping the utilization of the processors under a relatively high usage.

**Mean flow time** $(M/M/N_a/N_p)$. The average flow time $W$ is the key factor to evaluate the performance of the system. The theoretical calculation of the mean flow time of each event handling can be calculated as follows, based on the traditional $M/M/c$ queue model. Define $\Pi_W$ to be the probability that an event request has to wait. So $\Pi_W$ is the sum of the probabilities that this system contains $i$ requests $p_i$, where $i \geq c$:

$$\Pi_W = p_c + p_{c+1} + p_{c+2} + ...$$

$$= \frac{p_c}{1 - \rho} = \frac{(cp)^c}{c!} \left( (1 - \rho) \sum_{n=0}^{c-1} \frac{(cp)^n}{n!} + \frac{(cp)^c}{c!} \right)^{-1}$$

Substitute $c$ with $N_a$, then:

$$\Pi_W = \frac{(N_a \rho)^{N_a}}{N_a!} \left( (1 - \rho) \sum_{n=0}^{N_a-1} \frac{(N_a \rho)^n}{n!} + \frac{(N_a \rho)^{N_a}}{N_a!} \right)^{-1}$$

(2)

Then the mean waiting queue length $L_q$ is:

$$L_q = \sum_{n=0}^{\infty} np_{N_a+n} = \Pi_W \cdot \frac{\rho}{1 - \rho}$$

Then the mean queue length $L$ is:
\[ L = L_q + \frac{\lambda}{\mu e(N_p)} = \Pi_W \cdot \frac{\rho}{1 - \rho} + \frac{\lambda}{\mu e(N_p)} \]

According to Little’s Law [133], the average waiting time \( W_q \) is:

\[ W_q = \frac{L_q}{\lambda} = \Pi_W \cdot \frac{1}{1 - \rho} \cdot \frac{1}{\mu N_a e(N_p)} \]

Then the mean flow time of \( W \) in \( M/M/N_a/N_p \) is [134]:

\[ W_{M/M/N_a/N_p} = W_q + \frac{1}{\mu e(N_p)} = \Pi_W \cdot \frac{1}{1 - \rho} \cdot \frac{1}{\mu N_a e(N_p)} + \frac{1}{\mu e(N_p)} \]  

(3)

**Mean flow time** \((M/D/N_a/N_p)\). This section presents the mean flow time when the service time is deterministic. In Queue Theory, it is not difficult to calculate exact answers for the \( M/D/c \) system, but the calculations are more burdensome than for the corresponding \( M/M/c \) system [135]. Therefore, in order to get the mean flow time for \( M/D/N_a/N_p \), apply the following approximation proposed in [136]:

\[ L_{M/D/c}^{app} = \frac{1}{2} \left[ 1 + (1 + \rho)(c - 1) \frac{\sqrt{4 + 5c - 2}}{16\rho} \right] L_{M/M/c} \]

Applying Little’s Law, and substituting \( c \) and \( \rho \), the mean flow time in \( M/D/N_a/N_p \) is as follows:

\[ W_{M/D/N_a/N_p} = L_{M/D/c}^{app} \cdot \frac{1}{\lambda} = \frac{1}{2} \left[ 1 + (1 + \frac{\lambda}{\mu N_a e(N_p)}) (N_a - 1) \left( \frac{\sqrt{4 + 5N_a - 2}}{16\lambda} \right) L_{M/M/c} \right] \]

(4)

**Discussion of the model**

For an event-driven system, if utilization \( \rho \) is greater than 1, events are arriving faster than they can be handled, so the event queue will grow without bound. In another word, an event-driven system is not in a steady state if \( \rho > 1 \). Therefore, this section only discusses the systems that with steady state \((0 \leq \rho \leq 1)\).

**Relationship between flow time and processor utility**

It should be noticed that the value of \( \rho \) is subjected to \( \mu \) and \( \lambda \) (and \( N_a, N_p \) in a parallel system). In the following discussions, in order to explicitly demonstrate the relationships between \( \rho \) and other performance factors (e.g. \( W \)), \( \rho \) is shown as an independent variable as x-axis, but essentially it was \( \lambda / \mu N_a N_p \) changed.
Performance modeling of event-driven programs with OpenMP

Figure 35: The theoretical relationship between processor utilization $\rho$ and the mean event-handling flow time $W$ in an event-driven system.

According to equation 3, Figure 35 shows a plot of the relationship between $W$ and $\rho$ in the $M/M/N_a/N_p$ and $M/D/N_a/N_p$ model. If $N_a$ and $N_p$ are fixed values, it is easily found that the average flow time $W$ increases rapidly when utilization $\rho$ is above 80%. This leads to a dilemma that the system cannot reach both very high utilization and high performance, i.e low average flow time. If the event arrival rate is known and fixed, even though increasing $N_a$ and $N_p$ can reduce the mean request flow time $W$, it is unwise to distribute very large numbers of $N_a$ and $N_p$ since it causes a low processor utilization. In practice, keeping the utilization $\rho$ between 70% to 80% is considered a good operational level, without degrading much performance.

Distribution of asynchronization and parallelization

In a system with a fixed number of processors, it is interesting to study what has a better impact on the average flow time: the number of asynchronous workers ($N_a$) or the number of parallel works in an handling function($N_p$). Figure 36 and Figure 37 show the results when the total number of processors is fixed as 16, and how merely increasing $N_a$ or $N_p$ effects the average flow time (assume parallelism with idea speedup $\epsilon(N_p) = N_p$). So in this scenario where the event handlers are perfectly parallelizable, increasing the parallelization scale is a better choice to decrease the average flow time $W$.

Figure 38 studies this observation from a different perspective that accounts for real (non-ideal) parallelization speedups, i.e. $\epsilon(N_p) < N_p$. In this figure, the x-axis represents the number of threads that are used to parallelize the handling functions ($N_p$), and the
Figure 36: The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 0.8, \mu = 1$).

Figure 37: The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 10, \mu = 12$).
Figure 38: Minimum speedups $\epsilon(N_p)$ should be achieved that make the same $W$ when the scale is applied to asynchronization.

The y-axis indicates the minimum parallelization speedup $\epsilon(N_p)$ needed to achieve the same average flow time as using the same number of processors for asynchronization instead (i.e. as servers). The curve is only affected by $\rho$, and if the speedup is higher than a specific value, parallelization is always superior to asynchronization. For example, in $M/M/N_a/N_p$ with $\rho = 80\%$, this speedup value is close to 5; Whereas in $M/D/N_a/N_p$ when $\rho = 80\%$, the minimum speedup value is around 2.5.

EVALUATION

This sections presents a new evaluation of the parallelized event-driven programming approach using the virtual target extension for OpenMP. By doing this, it reflects back on the performance model developed in Section 5.2. The experiment shows how the event handling flow times are reduced by using a customized combination of parallelization and asynchronization.

This section evaluates the performance of a parallelized event-handling application. To do so a benchmark is implemented that simulates the behavior of a computational server, which provides several web services. In this synthetic application, the services are the following realistic computational kernels: Crypt, Monte Carlo, Series and Ray Tracer, which are selected from Java Grande Forum Benchmarks [137].

Every time a client requests a computation, the corresponding request data is sent via web socket. When the server receives a computation request, it queues its related handler function until resources are available for execution. When completed, the related data
is sent back to the client. For simplicity, when an event handler is queued, it cannot be canceled from the queue.

This benchmark application was implemented in Java. The system environment for execution is a 64-core AMD Opteron Processor 6272 SMP machine with 256 GB memory, and Java 1.8.0_101 HotSpot 64-Bit Server VM.

The sequential running times of all computational kernels are initially measured. The random creation of requests is governed by a Poisson Process, with specific parameters for each of the four kernels. The parameter values of each kernel are listed in Table 8.

Use the notation $P_x A_y$ in the presentation of the results, where $x$ corresponds to $N_p$, that is the number of threads used to parallelize a kernel; and $y$ corresponds to $N_a$, that is the number of threads used as asynchronous workers. If $y$ is specified as a list of four numbers $(y_1, y_2, y_3, y_4)$ then the asynchronous workers are separated into groups, where each number corresponds to the asynchronous workers for each of the four kernels. $N_a$ is then given by $N_a = y_1 + y_2 + y_3 + y_4$.

Adjust asynchronization to decrease queue time

In the first experiment, parallelism is not used, but the asynchronization varies. In this section, the asynchronous versions are implemented in two different ways. First, it uses traditional OpenMP task directive to offload event handler executions to the parallel region thread pool OpenMP64, then there are totally 64 asynchronous works used ($P1A64$). This approach can be used here because the EDT can be adjusted in a synthetic application (which is not always possible, see discussion in Section 4.3.4). Second, in another implementation virtual targets are used to offload different handlers to different virtual targets, and the thread pool sizes are partitioned in two different ways ($P1A(6,13,17,28)$ and $P1A(4,3,10,7)$).

Figure 39 compares the performance of the three approaches.
Figure 39: The mean flow times (ms) of four kernels implemented by three different asynchronization scale distributions.

**OpenMP64(P1A64):** In this approach, a global OpenMP parallel region uses all processors in a single 64-thread task pool and each request is executed as an OpenMP task. The performance is not as good as expected. For three out of four kernels, the mean flow times are drastically longer than the kernel sequential running times (the mean flow time of every kernel takes 122%, 276%, 215%, 191% of its sequential running time respectively, this is referred to as the mean stretch), which means each handler is taking a long time queuing.

**P1A(6,13,17,28):** This approach is implemented according to the virtual target concept, where four virtual targets are used with different thread-pool sizes. The total 64 asynchronous workers are distributed to four kernel handlers according to their sizes of sequential running times. Therefore, every kernel handler gains the proportion of 9%, 20%, 27%, 44% of the total asynchronous workers. Under this distribution, the results show a better performance than P1A64, although the total number of used threads does not change.

**P1A(4,3,10,7):** In this approach, in order to ensure high utilization, the number of asynchronous workers for each kernel handler is calculated by \( N_a = \lceil \frac{1}{\rho \lambda} \rceil \) (according to Equation 1 where \( \rho = 0.8 \)). Therefore, a total number of 24 asynchronous workers are distributed to four kernel handlers as A(4,3,10,7). The results show a very close performance comparing to P1A(6,12,18,28) but only 24 processors are used.

From this experiment, it can be concluded that offloading event handling tasks based on their run times can effectively decrease the mean flow times for handlers. Moreover, according to the performance model developed in Section 5.2, a succinct use of processors can be achieved without significantly degrading the performance.
Using both asynchronization and parallelization to decrease flow time

This section repeats the previous experiments, but also adjusts the parallelization, i.e. use more than a single thread for each kernel. The parallelization of each kernel is done using the traditional OpenMP for directive. The mean flow time of each type of event handler can be further reduced, therefore increasing the throughput of the server. Figure 40 depicts the case where all event handlers share a common asynchronous thread pool and each handler is parallelized with same number of threads. The results reveal that even with the same number of total processors, the different distributions between $N_a$ and $N_p$ can drastically influence the performance and that the best average flow time depends on the application. For example, it can be seen from the figure that for the MonteCarlo kernel, P6A14 is the best distribution, whereas for Raytracer P16A4 is better. It is also clear that using only asynchronisation (left) or only parallelism (right) are clearly inferior solutions.

![Figure 40: 64-processor distribution and its mean flow time (ms), categorized by different kernel handlers.](image)

Then the experiment tries to redistribute the processors for a better performance. Compared with the OpenMP tasking concept, virtual target enables programmers to have a more flexible processor distribution among a group of handlers, and its fine-grained thread pool control has the potential to gain a better performance. As instructed by Section 5.2.4.2, increasing parallelization scale can achieve better performance than applying asynchronization scale.

Figure 41 shows by the speedups of the four kernels and the speedup threshold curve that parallelization is worthwhile when $\rho = 80\%$. In this figure, as long as the speedup of the kernel is higher than the speedup threshold, increasing the parallelization scale...
Figure 41: Parallel speedup of four kernels and the minimum speedup threshold for $\rho = 0.8$.

is more worthwhile than increasing the asynchronization scale. As discussed in Section
5.2.4.2, in the $M/D/N_a/N_p$ model with $\rho = 80\%$, the speedup threshold for using
parallelization rather than asynchronization is relatively low. For every kernel, if the
kernel can be parallelized and the parallelism speedup is above 2.5, using parallelization
always outperforms asynchronization. As Figure 41 indicates the kernel Series and
RayTracer have better parallelism scale, allocating more processors to parallelize these
two kernels can lead to short average flow times.

Then the following strategy should be applied: Four virtual targets are used and
each virtual target uses one asynchronous worker. This means every type of kernel is
offloaded to its unique asynchronous worker. The remaining threads, are distributed to
parallelize the handlers. Figure 42 verifies this using the corresponding thread distribu-
tion.

Figure 42: Performance comparison of two types of distribution between asynchronization and
parallelization with total amount of 64 processors.
**P(6,13,17,28)A(1,1,1,1):** For every event handler, only one asynchronous worker is assigned, the remaining threads (processors) are distributed to parallelize event handlers. Compared to the previous distribution shown in Section 5.3.0.1, the results show that the new distribution has a better performance than the purely asynchronous distribution (left in Figure 42).

![Figure 43: Performance comparison of two types of distribution between asynchronization and parallelization with total amount of 24 processors.](image)

According to the speedups in Figure 41, some kernels do not scale well with increasing \( N_p \), the performance is not improved (e.g. Crypt, MonteCarlo).

**P(4,3,10,7)A(1,1,1,1):** Therefore, another distribution is tried. Figure 42 shows another distribution where a total of 24 processors are used. Compared to the previous distribution, using less threads actually further improves performance, in fact this distribution has the best performance throughout the experiment.

The experiment demonstrates that the properties of the event model (e.g. arrival distribution, service/execution time distribution), and the properties of the event handlers (e.g. sequential processing time, parallel scalability) are all factors that should be taken into consideration when allocating processors to achieve a better performance.

**SUMMARY**

This chapter studied how the performance of an event-driven system can be influenced by applying different parallelization and asynchronization scales. A mathematical model based on queue theory was proposed to model the performance of event-driven programs. The theoretical analysis reveals the fact that parallelism is very useful to reduce the mean flow time of the event handlers and threadsprocessors should be distributed correspondingly if the event handlers have the potential to be efficiently par-
allelized. The mathematical analysis was confirmed by the experimental evaluations, in which the program is implemented employing a previously proposed OpenMP virtual target extension. The experiments also show the advantage of the virtual target concept because it endows programmers with the flexibility to tune the performance according to the properties of event handlers. According to the mathematical model proposed in this chapter, programmers are able to tune the performance when implementing a parallel event-driven system. However, a better solution is endowing Pyjama the feature that is able to automatically adjust asynchronization and parallelization scale during the runtime. Some parameters can be set up using environmental variables before starting the program. Then the runtime dynamically distributes processors according to the instant event arrival rates. This can be a future work of this study.
Some recent researches on Java for High Performance Computing (HPC) shows that Java is a competent language for the HPC area (e.g. [138, 139, 140]). However, the performance and scalability of Java compared to other native languages is still arcane and hard to compare. This chapter proposes a benchmark suite to evaluate the performance and scalability of Java OpenMP implementations. The benchmark suite is based on the Java port of PolyBench, a Polyhedral Benchmark suite. PolyBench is selected instead of other existing benchmarks, like Java Grande Forum (JGF) benchmark, as it allows programmers to run and use the OpenMP C version as a performance and scalability reference. Further, PolyBench was conceived as a benchmark suite to analyze the optimization capabilities of compilers. It is interesting to study these capabilities in the OpenMP context of a dynamically compiled language like Java in comparison to the statically compiled C. In the benchmark suite, two Java OpenMP implementations are applied, Pyjama and JOMP, and compared with C code compiled by GCC, optimized and unoptimized. The sometimes surprising and unexpected results shed light on the appropriateness of Java as an OpenMP platform, the areas for improvement and the usefulness of this benchmark suite.

The contributions of this chapter are as follows:

- A complete benchmark suite is presented that can be used to evaluate the performance of Pyjama on traditional OpenMP parallelization constructs.

- The performance and scalability of Java OpenMP and C OpenMP is compared, and the result reveals that Java is a competent language to achieve high performance parallelization, but it has a higher run time threshold than C for any meaningful parallelism.
This chapter uses material in a published paper in the Proceedings of the 2016 12th International Workshop on OpenMP [141].

The remainder of this chapter is listed as follows: Section 6.1 introduces the difference between Java OpenMP and the standard C/C++ OpenMP. Section 6.2 discusses the methodology and guidelines of the benchmark suite. Section 6.3 describes the experimental setup, with performance results presented in Section 6.4. Section 6.5 presents related work before concluding in Section 6.6.

INTRODUCTION

OpenMP has long served the traditional parallel computing community with its constant adaption in response to the newest hardware technologies, as well as promoting programmer productivity. In recognition of Java’s popularity in the programming arena, numerous efforts have proposed a Java version of OpenMP [46, 47, 53, 48, 49]. As future efforts continue in this direction, having a standardized empirical evaluation grows in importance. This will provide the researchers and developers of Java OpenMP technologies with an accepted approach to present their development. PolyBench [142] is such a respected benchmark suite that helps developers to consistently validate their compilers, written in C. While one aspect of the focus here is that with a Java PolyBench port, more important is the characteristic differences between a statically compiled language and a dynamically compiled language. This will set the scene for better insight into Java’s future aptitude as a viable OpenMP language.

The optimization mechanisms of Java and C are very different. The portability qualities of Java have contributed to its popularity in mainstream software development. Java source code is compiled to Java bytecode, which is in turn interpreted by the Java Virtual Machine (JVM) using Just-In-Time (JIT) compilation. As a consequence, a Java program’s optimization is undertaken at runtime by the JVM to pursue better performance. This mechanism is very different from the execution of C/C++ programs, where optimization occurs in the compilation stage. Although the performance of a benchmark will depend on many factors including hardware architecture, operating system, and specific benchmark characteristics, it is still beneficial to understand how the optimization differences of these languages differ in the context of a parallel programming environment.
POLYBENCH FOR JAVA OPENMP

Even though it is reasonable that the performance of Java is not as competent as C/C++, referencing the performance of C/C++ OpenMP could help better understand Java OpenMP performance. Therefore, the methodology of this work is converting pre-existing C/C++ OpenMP benchmark cases to Java code implementing OpenMP. The major benchmark suite selected in this work is PolyBench, a benchmark suite developed for the polyhedral community that is used by many members of the community [143]. Because most of the benchmarks focus on linear algebra, data mining, medley and stencil problems that are parallelization candidates, PolyBench has drawn the attention of the parallel processing community. The reason PolyBench is chosen is because the benchmarks simulate real world problems, and all of them contain massive numerical operations and iterations. The performance of these kernels can be improved with parallelization, especially as they include large amounts of for-loops that are perfect candidates for the OpenMP programming interface.

There is a total of 30 available benchmark cases in the PolyBench suite. The original kernels are written in C with OpenMP pragmas, which may then be compiled by OpenMP-supported C/C++ compilers. Each kernel is then ported to Java, with OpenMP directives injected that conform to the requirements of the available Java OpenMP compilers. With the object-oriented nature of Java, each Java kernel exists as an independent Java class. In order to compare the performance under the same criteria, every Java implementation uses exactly the same directives and schedule clauses as its paired C implementation counterpart. The kernel naming follows the following convention: the first letter “P” stands for PolyBench while the second letter “L”, “D”, “M” or “S” stands for linear-algebra, data-mining, medley or stencils respectively, followed by the original benchmark name in PolyBench/C.

Table 9 and Table 10 list all the ported and parallelized benchmarks with their a description.

EVALUATION

This section details the experimental setup and the results are presented and discussed in the next section.
Table 9: List of implemented PolyBench benchmarks, with their sequential execution times (ms).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Java</td>
</tr>
<tr>
<td>PL2mm</td>
<td>Multiplication of 2 matrix</td>
<td>27029</td>
</tr>
<tr>
<td>PL3mm</td>
<td>Multiplication of 3 matrix</td>
<td>40199</td>
</tr>
<tr>
<td>PSadi</td>
<td>Alternating direction implicit solver</td>
<td>3787</td>
</tr>
<tr>
<td>PLatax</td>
<td>Matrix transpose and vector multiplication</td>
<td>112</td>
</tr>
<tr>
<td>PLbicg</td>
<td>BiCG sub kernel of BiCGStab linear solver</td>
<td>106</td>
</tr>
<tr>
<td>PLcholesky</td>
<td>Cholesky decomposition</td>
<td>387</td>
</tr>
<tr>
<td>PDcorrelation</td>
<td>Correlation computation</td>
<td>6621</td>
</tr>
<tr>
<td>PDcovariance</td>
<td>Covariance computation</td>
<td>6577</td>
</tr>
<tr>
<td>PLdoitgen</td>
<td>Multi-resolution adaptive numerical analysis</td>
<td>677</td>
</tr>
<tr>
<td>PLdurbin</td>
<td>Toeplitz system solver</td>
<td>574</td>
</tr>
<tr>
<td>PLdynprog</td>
<td>2-D Dynamic programming</td>
<td>1073</td>
</tr>
<tr>
<td>PSfdtd_2d</td>
<td>2-D finite different time domain kernel</td>
<td>1298</td>
</tr>
<tr>
<td>PSfdtd_apml</td>
<td>FDTD using anisotropic perfectly matched layer</td>
<td>2705</td>
</tr>
<tr>
<td>PMfloydWarshall</td>
<td>Floyd–Warshall algorithm</td>
<td>5138</td>
</tr>
<tr>
<td>PLgemm</td>
<td>Matrix multiplication and addition</td>
<td>13138</td>
</tr>
</tbody>
</table>
Table 10: List of implemented PolyBench benchmarks, with their sequential execution times (ms) (continued).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Java</td>
</tr>
<tr>
<td>PLgemver</td>
<td>Vector multiplication and matrix addition</td>
<td>479</td>
</tr>
<tr>
<td>PLgesummv</td>
<td>Scalar, vector and matrix multiplication</td>
<td>130</td>
</tr>
<tr>
<td>PLgramschmidt</td>
<td>Gram-Schmidt decomposition</td>
<td>1780</td>
</tr>
<tr>
<td>PSjacobi_1d_imper</td>
<td>1-D Jacobi stencil computation</td>
<td>38</td>
</tr>
<tr>
<td>PSjacobi_2d_imper</td>
<td>2-D Jacobi stencil computation</td>
<td>331</td>
</tr>
<tr>
<td>PLlu</td>
<td>LU composition</td>
<td>1277</td>
</tr>
<tr>
<td>PLLudcmp</td>
<td>LU decomposition</td>
<td>3435</td>
</tr>
<tr>
<td>PLmvnt</td>
<td>Matrix vector product and transpose</td>
<td>384</td>
</tr>
<tr>
<td>PMreg_detect</td>
<td>2-D image processing</td>
<td>86</td>
</tr>
<tr>
<td>PSseidel_2d</td>
<td>2-D Seidel stencil computation</td>
<td>508</td>
</tr>
<tr>
<td>PLsymm</td>
<td>Symmetric matrix-multiply</td>
<td>17547</td>
</tr>
<tr>
<td>PLsyr2k</td>
<td>Symmetric rank-2k operations</td>
<td>5981</td>
</tr>
<tr>
<td>PLsyrk</td>
<td>Symmetric rank-k operations</td>
<td>2902</td>
</tr>
<tr>
<td>PLtrisolv</td>
<td>Triangular solver</td>
<td>50</td>
</tr>
<tr>
<td>PLtrmm</td>
<td>Triangular matrix-multiply</td>
<td>1393</td>
</tr>
</tbody>
</table>
Table 11: Four types of compilation of OpenMP code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-o0</td>
<td>gcc -O0 -fopenmp -DPOLYBENCH_TIME -I benchmark.c -o benchmark</td>
</tr>
<tr>
<td>C-o3</td>
<td>gcc -O3 -fopenmp -DPOLYBENCH_TIME -I benchmark.c -o benchmark</td>
</tr>
<tr>
<td>Jomp</td>
<td>java -cp jomp1.0b.jar jomp.compiler.Jomp benchmark.jomp</td>
</tr>
<tr>
<td>Pyjama</td>
<td>java -jar Pyjama-v1.5.4.jar benchmark.pj</td>
</tr>
</tbody>
</table>

Compilation

The OpenMP C code can be compiled by any C compiler which supports OpenMP. In this experiment, GCC v4.8.4 is used to compile all C benchmarks. With regard to Java code, this experiment uses two different Java implementations of OpenMP – Pyjama (v1.5.4b) and Jomp (v1.0b) to parallelize the Java OpenMP benchmarks.

The compilation details are listed in Table 11. Since the optimization of C code only happens at the compilation stage, two different optimization levels were used to compile the C code. More specifically, flag -O0 indicates no optimization at all. Flag -O3 triggers very salient static code optimization to improve the efficiency of the C program, including vectorization.

In contrast, the compilation from Java source code to Java bytecode is not given any optimization option and the virtual machine does so many just-in-time optimization at runtime.

Execution

All the benchmarks are executed on a 16-core 2.4GHz SMP machine (4 quad-core Intel Xeons E7340) with 64 GB memory, running under Ubuntu Linux 12.04 LTS and Java HotSpot 64-Bit Server VM (1.8.0_66) is used. Regarding Java benchmark cases, as a matter of fairness, they do not apply any deliberate warm-ups to make Java Virtual Machine discover hot spots, which could lead to better performance. For every benchmark, the same data set is used for all implementations.

The sequential execution times are also listed in Table 9 and Table 10, with the standard data set, as specific by PolyBench.
It can be observed from the tables that for 29/30 benchmark cases, the sequential execution time of Java version is faster than the unoptimized C version, and even 8/30 benchmarks are faster than the optimized C version, which is somewhat unexpected. According to the execution time of Java, all the benchmark cases are categorized into three groups. There are 8/30 benchmark cases whose execution times are longer than 5s, and there are 9/30 cases whose execution times are between 1s and 5s. All the remaining 13/30 cases have less than 1s execution times.

RESULTS

The absolute running times of all implemented benchmark cases are recorded over different numbers of processors. The speedups of each implementation are calculated according to the execution times. The first observation the experiment made is that the absolute runtimes have a significant impact on the scalability of the Java implementations; therefore this section organizes the discussion along the benchmark groups (>5s, 1s–5s, <1s) in the following:

Long runtimes

For the benchmarks where Java’s execution time is longer than 5s, Java shows very good performance and scalability. It is surprising that in some cases, the Java implementations have very impressive execution times and even better than the optimized C versions. These benchmark cases include PDcorrelation, PL2mm, PL3mm, PLgemm and PLsymm. For these 5 benchmarks, the Java versions are significantly faster than the C versions, which is more than twice the speed with any number of threads. The scalability of the Java versions is also comparable to the C versions and even better than C in some cases. Most of these benchmarks are linear algebra problems and the computational kernels involve large numbers of manipulations on matrices. A nested loop without any dependency is the best candidate to be parallelized and reaches good speedup. At the same time, the long running time could be an important factor in that the JVM discovers the hot spots and optimizes the execution.

Figure 44 shows three selected benchmark cases which have this type of characteristic. It can be seen that both Jomp and Pyjama have faster speeds than both C versions with regard to execution time. When it comes to scalability, all four implementations have
Figure 44: Selected benchmark cases (PL2mm, PL3mm and PLsymm) where Java implementations have good performance and scalability (comparing to C).
Figure 45: Selected benchmark cases (PLtrmm, PLsyrk and PSfdtd_2d) where Java implementations have comparable performance with C versions, but scalability is not very good.

similar speedups. For most of the time, Java implementations (either Jomp or Pyjama) lead among the four implementations.

Medium runtimes

For the second category, in which the execution time of Java is between 1s and 5s, Java OpenMP also shows some performance improvements but not in a prominent way. For many of the benchmark cases in this group, the performance of Java versions is faster than the unoptimized C version, but cannot compete with the optimized C version. These benchmark cases include PLtrmm, PLlu, and PLsyrk.
Figure 46: Selected benchmark cases (PLtrisolv, PLatax, and PLmvt) where Java implementations have bad performance and scalability.

Figure 45 illustrates some examples of this situation. Taking PLtrmm for example, the execution times of Jomp and Pyjama nearly overlay with each other, which indicates they have very similar performance. Even though most of the time, the execution of Java versions is faster than the unoptimized C, the optimized C version is always better than the other three versions. When it comes to speedup, both C versions have better scalability than Java versions. It should be noted that even though Java versions are faster than the unoptimized C version, the scalability is poor for both Jomp and Pyjama. For instance, the speedup of Java versions never reaches three for benchmarks PLtrmm and PSfdtd_2d. In contrast, both C versions have better speedup than Java.
**Short runtimes**

As for the last group of benchmark cases, where the execution time of Java is lower than 1s, both Jomp and Pyjama do not have very good execution times, and even worse, Java has a longer execution time when the number of threads is increased. This type of benchmark indicates that for some situations, Java OpenMP parallelization can even degrade the performance.

Figure 46 selects PLatax, PSfdtd_ampl and PLmvt to demonstrate this type of benchmark results. Besides the long execution time of Java, the speedups also show the bad scalability of Java parallelization of these benchmark cases. It seems obvious that the Java execution times are too short to benefit from parallelization, where C as an OpenMP language is still able to achieve speedup. Lower parallelization overhead sets the threshold for useful parallelization lower for C.

**Other Observations**

For most of the cases, even though the execution time is not always better than Java and the optimized and unoptimized C version always has better scalability than the three other versions. On the other hand, although the Java versions outperform in some of the benchmarks, they do not show very good speedups with regard to scalability.

**Non-scalable benchmark cases**

There are three benchmark cases (PLdynprog, PSjacobi_1d_imper and PMerg_detect) that cannot benefit from both C and Java parallelisation. When the number of threads increases, the execution time gets even slower than sequential. This may be due to the nature of these computational kernels and that they are not suitable for OpenMP parallelisation.

Figure 47 shows the results of PLdynprog and PMreg_detect. Nearly all four versions show bad performance and the execution time even grows when the number of threads increases. Even though the execution time of PLdynprog is long enough (around 1-3s), when the number of threads increases, two Java versions and the optimized C version degrade the performance, and the unoptimized C only gains very low speedup (up to 2).
Figure 47: Benchmark cases (PLdynprog and PMreg_detect) show their unscalability of parallelisation.

RELATED WORK

This section discusses related work regarding to Java OpenMP efforts and benchmarks.

Java benchmarks

The SPECjbb benchmark suite [144] is typically used to evaluate the server-side Java performance in different systems. At the same time, numerous Java benchmarks have been developed to test the performance of the JVM, e.g. Decapo [145], Java Grande Forum Benchmarks [137, 79]. Some benchmarks specially focus on Java parallelization [99] or multi-threading [146].

Parallel processing benchmarks

There are also many benchmark suites which are designed to evaluate the performance of parallel processing although they are not written in Java. For example, SEPCComp [147] is an OpenMP code based benchmark suite which is used to evaluate high-performance multi-core systems. The PARSEC benchmark suite [148, 149] implements a series of state-of-art multi-threading applications for evaluating Chip-Multiprocessors.
SUMMARY

This chapter proposed a new benchmark suite to evaluate the performance of OpenMP implementations of Java. It is based on the port of PolyBench to Java, augmented with (Java) OpenMP directives. The choice for this benchmark suite was driven by two criteria: to have an equivalent C implementation that can serve as a performance and scalability reference; and to use code with varying optimization potential for compilers.

With this benchmark suite to analyze the appropriateness of Java as an OpenMP platform, an experiment was carried out to study applying PolyBench to the OpenMP Java implementations Pyjama and Jomp and comparing with C code compiled by GCC. An observation was found that the runtime of some of the benchmarks is too small to allow efficient parallelization, especially for Java. While this can be seen as a deficit of this benchmark suite, it revealed that Java as an OpenMP platform has a significantly higher parallelization threshold than C. On the other hand, when the runtimes were long, OpenMP under Java did not only show similar scalability as C, it was surprisingly competitive in regards to the absolute runtime. The difference between the two Java implementations was relatively small with no clear winner, showing similar behavior at both ends of the spectrum. Overall, the experiments showed that the proposed PolyBench based benchmark suite is an effective tool to analyze Java OpenMP performance.
CONCLUSIONS

As one of the most popular parallel processing frameworks, OpenMP has proved to be a successful programming model for shared-memory architectures. An evolution over two decades makes OpenMP more and more powerful and flexible to parallelize various applications on different shared-memory machines. This research focuses on its unique concern: Combining OpenMP with object-oriented programming concepts and enabling the use of OpenMP in object-oriented languages more productive and less error-prone. The ultimate aim of this research is to develop new tools to improve programmers’ productivity and efficiency on object-oriented OpenMP parallel programs. New concepts and ideas have been proposed and implemented on project Pyjama – a Java implementation of OpenMP-like directives and runtime routines.

In languages such as Java, the concept of exception handling is of high importance and it has been an integral part from its first release. This error recovery model is not officially integrated into the OpenMP specification, so the challenges lie on supporting exception-based error recovery in OpenMP and extending both semantic specifications and related runtime support. This work proposed a systematic mechanism for exception handling with the co-use of OpenMP directives, and implemented it on a Java version of OpenMP. The concept of exception handling with OpenMP directives has been formalized and categorized. Along with the support of exception handling, a flexible approach to thread cancellation is also proposed to support customized parallel control flows. The runtime support and its implementation issues are discussed and the evaluations presented. The new approach is proven to be a more elegant coding style which increases the parallel development efficiency and software robustness whereas there is no prominent overhead introduced.
The event-driven programming pattern is pervasive in a wide range of modern software developments. Unfortunately, it is not easy to achieve good performance and responsiveness when developing event-driven applications. This work proposed a hybrid model for the combination of asynchronization and parallelization, as an extension of OpenMP to facilitate event-driven programs, especially for GUI applications. By using this programming model, the sequential programming logic can be easily parallelized for better responsiveness and execution performance. By stressing concurrency, the distinctions between the OpenMP task concept and virtual target are discussed, and it shows the advantages of using the virtual target concept. The examples show the effectiveness of parallelizing different event-driven applications.

Together with the proposed model for event-driven programming, a theoretical analysis of event-driven programs is also presented. This performance model can be used to investigate the best combination of asynchronization and parallelization. The performance model and the experiments reveal that parallelism is very useful to reduce the mean flow time of the event handlers and threads/processors should be distributed correspondingly if the event handlers have the potential to be efficiently parallelized. Also, the experiments demonstrate that the virtual target programming model is more flexible than the OpenMP tasking model in event-driven programs.

Evaluations show that single-threaded event dispatching can be quickly upgraded to a higher performing multi-threaded event dispatching, by reducing event handling response time. Performance achieved by the proposed directive based approach is equal and often superior to manual implementations.

FUTURE WORKS

This section lists several possible directions as future works that build on this thesis.

Worksharing tuning

Pyjama has been supporting a rich arrange of loop parallelization, from traditional numerical and high-level for-each loops and iterator based loops. However, the performance is still subject to the proper use of scheduling clauses and chunk-size selections. An interesting future direction is if the parallelization tool is able to smartly tune the loop scheduling and automatically gain better performance with little programming
effort. This consideration is endowing the compiler and runtime ability to automatically apply the best parallel execution according to the features and attributes of the loop.

There is already some work related to the performance tuning and data locality. For example, TunedCnC [150] is a flexible tuning framework. In light of a separation of concern, tuning experts are able to specify how the application should be executed on a given architecture without necessarily knowing the parallel application and its data and control dependencies. Larsen et al. [151] developed an interactive compilation feedback system to guide programmers to modify their source code to outperform other approaches. Generating loops optimized for caches requires a compiler that has the ability to analyze the data flow and a runtime system to better exploit data locality [152].

With regards to Java, due to bytecode being executed on the Java Virtual Machine (JVM), the performance is highly influenced by the JVM making it more difficult to reason about performance. As has been shown in the Chapter 6, Java has a much higher threshold than C/C++ to show the benefit of parallelization. Studying the JVM is important in understanding how bytecode-level parallelism is executed. A clear understanding of virtual machine cache locality is a key factor to achieve good Java parallelization performance tuning.

Incorporating asynchronous I/O to virtual target

The virtual target concept demonstrates a new perspective of executors. Different executors can be assigned to cope with different types of computational loads. However, using this scenario is under the presumption that the application is computation-bound. For some applications, I/O could be the main bottleneck to performance [153]. One thread per I/O is not applicable since the burst of thread numbers could lead to a large performance degradation [154]. In light of this situation, multiplexing I/O is used to offer a highly scalable and performant polling interface. Multiplexing is usually an operating system level support, and both synchronous I/O event handling [112] and asynchronous I/O event handling [111] are already proposed. However, the use of multiplexing requires a high level of expert knowledge and system level coding experience. Even though Java already supports asynchronous and non-blocking I/O as part of the standard library, it requires the breakdown of original sequential code which makes the code logic difficult to analyze. Some efforts have tried to simplify the interfaces to
make this high performance model more usable. For example, AC [155] proposed a new C/C++ I/O interface which retains the sequential composable programming style of synchronous operations but achieves high-performance asynchronous I/O. The concept of asynchronous I/O can be perfectly integrated into the virtual target concept because of its asynchronous nature. The syntax remains consistent without any changes and the work involves an implementation of I/O virtual target.

An I/O virtual target is an abstraction of the I/O event multiplexer. All I/O related operations are offloaded to the I/O virtual target. By polling, the I/O virtual target is able to handle many I/Os with one or few threads. The programming logic can be still represented in a sequential manner, but the I/O performance can be greatly improved.

Software transactional memory

Standard OpenMP still uses traditional lock or atomic operations to cope with synchronization. As is well-known that improper locks may lead to performance issue and multiple locks cannot be synthesized. Transactional memory shows promise for making parallel programming easier [156]. In conforming with the philosophy of OpenMP, the use of transaction syntax does not alter the sequential code [157]. Even though the performance is still the biggest issue, at least it is an off-the-shelf solution for novice programmers to write correct parallel programs. Software transactions have already shown some reasonable speedup to sequential executions on few-core systems [158], which is perfectly suitable for desktop, mobile apps and other daily life software developments.

Adopting software transactional memory for OpenMP is an interesting topic. By using directives and related clauses, the conversion from sequential code to a transaction block can be of both simplicity and expressiveness. Pyjama has the potential to be an experimental platform to conduct software transactional memory experiments for Java.

Event-driven online optimization

Chapter 5 has already shown the distribution between parallelization scale and asynchronization scale can hugely influence the performance of a parallel event-driven system. However, performance tuning is based on the presumption that the arrival rates of event-handlers are known and consistent. The events may change their arrival rates. For an online event-driven system, how to dynamically adjust the parallelization
scale in the runtime to suit the corresponding arrival rates is worth exploring. The challenges lie on how to measure the dynamic arrival rates in the runtime and make decisions to reach the best performance. In addition, if the event handlers are I/O-bound, choosing the correct size of the I/O asynchronous workers also affects the performance [159]. All of these aspects are worthy of further research efforts.
OpenMP Application Program Interface (API) is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

Pyjama is an OpenMP Java language implementation, with new features and extensions and its unique concerns for object-oriented parallel application development. It supports Java Virtual Machine (JVM) based shared-memory parallel programming on all architectures, from Unix platforms to Windows NT platforms to Android mobile platforms.

DIRECTIVES

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A structured-block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom. For Pyjama, the directive starts with an `//#omp` identifier, which looks like a comment line in Java programming language. Once the source code with directives is compiled through the Pyjama compiler, the paralleled Java executable bytecode will be generated.

Parallel

The parallel construct forms a team of threads and starts parallel execution.

`//#omp parallel [clause [, clause] ...]`
clause:

if(scalar-expression)

num_threads(integer-expression)

default(none | shared)

private(list) firstprivate(list) shared(list)

reduction(operator | operation-function list) copyin(list)

Customized reduction operation:
Pyjama supports user defined reduction operation. Programmers can define their own reduction operation by implementing reduction methods.

A function declaration must comfort with <T> reductionFunctionName(<T> var1, var2).

Loop

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
// #omp for [clause [, clause] ...]
```

for-loops

clause:

private(list) firstprivate(list) lastprivate(list)

reduction(operator | operation-function list) copyin(list)

schedule(kind[, chunk-size])

ordered

nowait

where kind is one of the following:

- static: Iterations are divided into chunks of chunk size. Chunks are assigned to threads in the team in round-robin fashion in order of thread number.
- dynamic: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be distributed.

1 In Pyjama, in light of good object-oriented programming style, we discourage the use of private data clause. Users may define new variables inside parallel region when thread-private variables are required.
2 We eliminate threadprivate directive in Pyjama, since it breaks design rules of object-oriented design using global variables. So copying data clause is banned as well.
• **guided**: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated chunk size as chunks are scheduled.

• **auto**: The decision regarding scheduling is delegated to the compiler and/or runtime system.

• **runtime**: The schedule and chunk size are taken from the run-sched-var ICV.

**Sections**

The sections construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

```c
// #omp sections [clause [, clause] ...]
{
  [ // #omp section]
  structured-block
  [ // #omp section]
  structured-block
  ...}
clause:
  private(list) firstprivate(list) lastprivate(list)
  reduction(operator|operation-function list) copyin(list)
  schedule(kind[, chunk-size])
  nowait
```

**Single**

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```c
// #omp single [clause [, clause] ...]
structured-block
clause:
  private(list) copyprivate(list)
  firstprivate(list)
```
nowait

Parallel Loop

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

```c
// #omp parallel for [clause [, clause] ...]
```

for-loop

Any accepted by the parallel or for directives, except the nowait clause, with identical meanings and restrictions.

Parallel Sections

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other statements.

```c
// #omp parallel sections [clause [, clause] ...]
{
    [ // #omp section]
    structured-block
    [ // #omp section]
    structured-block
    ...
}
```

Any of the clauses accepted by the parallel or sections directives, except the nowait clause, with identical meanings and restrictions.

Master

The master construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

```c
// #omp master
```

structured-block
Critical

The critical construct restricts execution of the associated structured block to a single thread at a time.

```c
#pragma omp critical [(name)]
structured-block
```

Barrier

The barrier construct specifies an explicit barrier at the point at which the construct appears.

```c
#pragma omp barrier
```

Flush

The flush construct executes the OpenMP flush operation, which makes a thread’s temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```c
#pragma omp flush [(list)]
```

Ordered

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```c
#pragma omp ordered
structured-block
```

Freeguithread

The freeguithread directive is designed for GUI application, which makes the GUI EDT thread free from executing current code and enables it to process the next event/message.

```c
#pragma omp freeguithread [parallel [,for|sections [clause [, clause] ...]]]
```
GUI

The gui directive is designed for GUI application. The code block after the GUI directive will be executed in GUI EDT. Clause nowait makes the current thread directly execute the next part of the code instead of waiting for the EDT’s completion of this code block. This directive is usually used in freeguithread to notify the EDT thread to execute following block.

```c
// #omp gui
structured-block
class:
nowait
```

Cancel

The cancel directive invokes a cancellation of a specific parallel region, sections, parallel for-loop, taskgroup. A global exception is

```c
// #omp cancel construct-type-clause thread-affiliate-clause [if-clause] [throw-clause]
structured-block
```

where construct-type-clause is one of the following:

- parallel, sections, for, taskgroup

and thread-affiliate-clause is one of the following:

- global, local

and if-clause is:

```c
if(scalar-expression)
```

and throw-clause is:

```c
throw(expression-name)
```

Cancellation Point

The cancellation point directive defines an explicit cancellation checking point. It can be used in any position of the source code. If this directive is inside a specific OpenMP
construct block, when this corresponding region is trigger canceled, the execution of this region will stop at this point.

```cpp
    // #omp cancellation point construct-type-clause thread-affiliate-clause [if-clause] [throw-clause]
    structured-block

    where construct-type-clause is one of the following:
    parallel, sections, for, taskgroup, task
    and thread-affiliate-clause is one of the following:
    global, local
```

**Task**

The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```cpp
    // #omp task [clause [, clause] ...]
    structured-block

    clause:
    firstprivate(list) shared(list)
    if(scalar-expression)
    on_cancel(function-call-expression)
```

**Virtual Target**

The virtual target concept is designed for supporting an asynchronization model for event-driven programming patterns. For detailed usage, please refer to the publications of Pyjama.

```cpp
    // #omp target virtual [clause [, clause] ...]
    structured-block

    clause can be the following:
    asynchronous-property-clause
    data-handling-clause
    if-clause
    on-cancel-clause
```
where *asynchronous-property-clause* is one of the following:

**nowait name_as(name-tag) await**

where *data-handling-clause* is one of the following:

**firstprivate(list) shared(list)**

and *if-clause* is:

**if(scalar-expression)**

and *on-cancel-clause* is:

**on_cancel(function-call-expression)**

### Task Cancel

The task cancel directive defines an explicit task cancellation. If no *task-name* is followed, all the OpenMP tasks will be canceled. If this directive is followed with a *task-name*, all the tasks with this name will be canceled.

```
#pragma omp taskcancel [(task-name)] [if-clause]
structured-block
if-clause is:
if(scalar-expression)
```

### Task Wait

The task cancel directive defines a task waiting point. The current control flow will not proceed until the task/tasks are complete. If no *task-name* is followed, the current thread will wait for all the OpenMP tasks in the parallel region. If this directive is followed with a *task-name*, the current thread waits until all the tasks with this name are complete.

```
#pragma omp taskwait [(task-name)] [if-clause]
structured-block
if-clause is:
if(scalar-expression)
```

### Async

The async directive is designed to annotate the following function has the asynchronous nature.
Async Call

The async call directive is designed for invoking functions in an asynchronous way. Every function inside the following block will check if it was declared in the directive function declaration list. If yes, this function will be executed asynchronously according to the asynchronous-property-clause.

```c
#pragma omp async
function-declaration
```

### Async Call

```c
#pragma omp async-call asynchronous-property-clause(function-declaration [, function-declaration]...)
```

structured-block

where asynchronous-property-clause is one of the following:

- `nowait name_as(name-tag) await`

**RunTime Library Routines**

**Parallel Execution Environment Routines**

Execution environment routines affect and monitor threads, processors, and the parallel environment.

- `int Pyjama.omp_get_num_threads()`
  
  Returns the number of threads in the current team.

- `void Pyjama.omp_set_num_threads(int num_threads)`
  
  Affects the number of threads used for subsequent parallel regions that do not specify a num threads clause.

- `int Pyjama.omp_get_thread_num()`
  
  Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

- `boolean Pyjama.omp_in_parallel()`
  
  Returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false.
• int Pyjama.omp_get_max_threads()
  Returns maximum number of threads that could be used to form a new team using a parallel construct without a num threads clause.

• int Pyjama.omp_get_num_procs()
  Returns the number of processors available to the program.

• int Pyjama.omp_get_dynamic()
  Returns the value of the dyn-var ICV, determining whether dynamic adjustment of the number of threads is enabled or disabled.

• void Pyjama.omp_set_dynamic()
  Enables or disables dynamic adjustment of the number of threads available by setting the value of the dyn-var ICV.

• void Pyjama.omp_set_nested(boolean nested)
  Enables or disables nested parallelism, by setting the nest-var ICV.

• boolean Pyjama.omp_get_nested()
  Returns the value of the nest-var ICV, which determines if nested parallelism is enabled or disabled.

• boolean Pyjama.omp_in_parallel()
  Returns the boolean value if the routine call is inside a parallel region.

• int Pyjama.omp_cancellation()
  Returns the value of the cancel-var ICV, which controls the behavior of the cancel directive and cancellation point.

• int Pyjama.omp_set_schedule()
  This routine affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the run-shed-var ICV.

• int Pyjama.omp_get_schedule()
  Returns the schedule that is applied when the runtime schedule is used.

• int Pyjama.omp_get_thread_limit()
  Returns the maximum number of OpenMP threads available on the device.
Asynchronous Execution Environment Routines

Execution environment routines affect and support the functionality for event-driven offloading and asynchronization.

- **void Pyjama.omp_register_as_virtual_target(String targetName)**
  
  Register current thread as a virtual target, with name of targetName.

- **void Pyjama.omp_create_virtual_target(String targetName)**
  
  Create a single-thread virtual target, with name of targetName.

- **void Pyjama.omp_create_virtual_target(String targetName, int n)**
  
  Create an n thread virtual target, with name of targetName.

- **String Pyjama.omp_get_target_name()**
  
  If current thread belongs to a virtual target, return the target name. If not, return null.

- **void Pyjama.omp_set_platform(Platform platform)**
  
  Set current GUI programming framework, choosing from one of the supporting platforms: Android, JavaFX and Swing.

- **String Pyjama.omp_get_platform()**
  
  Return the platform identifier current program is using.

Timing Routines

Timing routines support a portable wall clock timer.

- **double Pyjama.omp_get_wtime()**
  
  Returns elapsed wall clock time in seconds.

- **double Pyjama.omp_get_wtick()**
  
  Returns the precision of the timer used by omp get Pyjama.omp_get_wtime().
Environment Variables

- OMP_SCHEDULE
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED
BIBLIOGRAPHY


