Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

REGULAR PAPER

Balancing Parallelization and Asynchronization in Event-Driven Programs with OpenMP

Xing Fan | Oliver Sinnen | Nasser Giacaman

Parallel and Reconfigurable Computing Lab
Department of Electrical and Computer Engineering, The University of Auckland, New Zealand

Correspondence
Nasser Giacaman Email:
n.giacaman@auckland.ac.nz

Summary

OpenMP is a popular multiprocessing interface to parallelize different domains of applications. 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 of nested parallelism with asynchronous processing. This new possibility raises the question 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. Then 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.

KEYWORDS:
OpenMP, event-driven, performance model, virtual target, concurrency

1 | INTRODUCTION

A wide range of modern applications are developed based on the event-driven model, ranging from mobile applications, desktop applications and web services. In general, these types of programs have an interactive nature, which means their execution is non-deterministic, 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 multi-threading applied, the EDT executes all the queued event handlers in a sequential manner. The 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.

In order to achieve a high performing event-driven system, the widely used solution is to offload computation requests to the background thread pool. This approach enables the EDT to handle multiple requests asynchronously, thereby the system's handling ability is enhanced and the system's overall throughput increases. When considering to use OpenMP to parallelize event-driven programs, OpenMP task parallelism is the most suitable approach instead of using the traditional OpenMP worksharing constructs `for` and `sections`. Task parallelism has been an important part of OpenMP programming model, since its initial release of version 3.0 [1]. OpenMP task parallelism enables programmers to handle irregular and asymmetrical 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 [2] and task-generating loops [3].

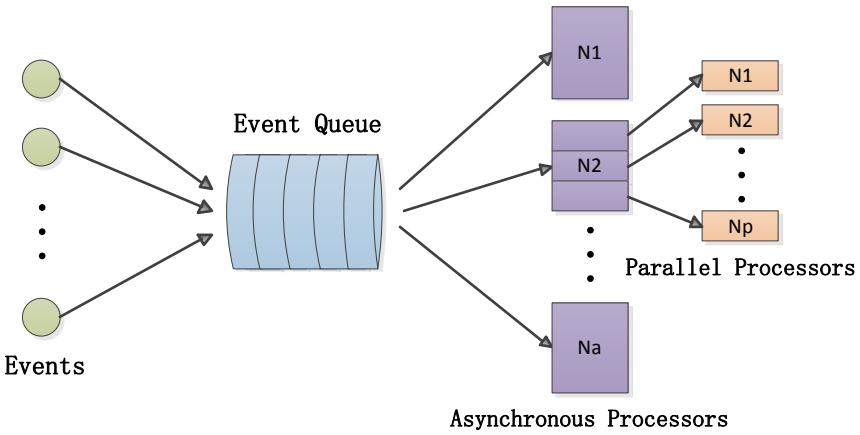


FIGURE 1 A parallel event dispatching model that has multiple asynchronous processors and parallel processors.

However, OpenMP tasking is mainly for task parallelization and its asynchronization is only confined within the parallel region, which itself is still synchronous. In comparison, an event-driven system demands that the computations be offloaded away from the EDT, hence cannot contain a parallel region for that. In this regard, it is *concurrency* rather than *task parallelism* that needs to be addressed. Unfortunately OpenMP does not have special constructs to deal with event-driven programming. Some efforts of using OpenMP for web services have been tried [4,5]. The experiences show that using traditional OpenMP directives to parallelize web service calls is possible, but it requires nontrivial programming efforts to achieve high level performance.

To overcome this, the concept of virtual target [6] has been proposed as a complementary part for OpenMP to facilitate event-driven parallel programming. We have verified this approach through our Java implementation of OpenMP, known as Pyjama [7]. With the proposed OpenMP extensions, a single-threaded Java program is easily converted to a multi-threaded version, supporting both parallel and asynchronous execution. In the virtual target approach, we enable *concurrency* (asynchronization), which detaches the control flow from the master thread, and offloads the execution block from the master thread to another asynchronous thread or thread pool.

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 1 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. In this paper we attempt to address these questions with a performance model and an experimental evaluation. An interesting fact outcome of this study is that parallelism is a more efficient way to increase the event-driven performance than asynchronization, if the event handlers have the potential to be parallelized. The theoretical model can help programmers make decisions on choosing the proper number 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.

In particular this paper presents the following contributions:

- A performance model for event-driven parallelization is proposed, derived from a queuing theory model. We illustrate and plot the influence of model parameters on performance. This model gives a theoretical reference for the benchmarks.
- A conceptual comparison between the virtual target and OpenMP tasking models is presented.
- An experimental evaluation is conducted that 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. The observations from these experiments give new insights into the interplay between asynchonization and parallelization.
  - We demonstrate experimentally that the responsiveness of an interactive GUI application is significantly improved using the tool LagHunter/LagAnalyzer [8 9] to analyze the latencies.

– We demonstrate that large reductions in mean flow-time can be achieved in an event-driven application by tuning the number threads/processors used for parallelism and asynchronization. The virtual target concept also allows to categorize different tasks, to be submitted to different thread pools according to their sizes, which thereby further minimizes the mean event handling flow time.

Parts of this work were published in a preliminary version at IWOMP 2017 [10]. In comparison, this paper presents a significantly extended and more detailed mathematical model, studies the implications more thoroughly and has a significantly extended experimental evaluation.

The structure of this paper is as follows: Section **??** describes the theoretical background of parallelizing event-driven programs and a mathematical model is presented to quantify the performance. Section **??** presents the OpenMP solutions to parallelize event-driven programs. In this section, the virtual target solution is presented and the comparison with OpenMP tasking is provided to show the advantage of using virtual target model. Section **??** presents the experiments performed and the results obtained, and it demonstrates the effectiveness of using the proposed programming model and performance model. Related work is described in Section **??**, before concluding in Section **??**.

2 | 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 we need some other measure to handle events in a system. For this reason this section establish a performance model based on queue theory.

2.1 | 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 $e \rightarrow \mathcal{F}$ in a system. When the event e happens, its event handling function $\mathcal{F}$ should be triggered. At the time the event e is triggered, there are potentially unprocessed event handlers in the event queue, and the set $\mathbf{F}$ 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 $\mathcal{F}$'s execution time (service time) t_S:

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

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.

2.2 | 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 we 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. η is the parallelization efficiency factor.

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

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:

$$t'_F = t_{UI} \mid \mathcal{T}_{edt} + t_F \mid \mathcal{T}_{worker}$$

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.

2.3 | 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, we employ the Kendall Notation that is used to describe queuing systems [11]. 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, we assume that this model has unlimited capacity of the queue, and the queuing principle is First In First Out (FIFO).

- A: Arrival process
  - M: 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

2.3.1 | Queue model with parallelism

Now we extend the $A/S/c$ model to integrate parallel execution of the jobs (event handlers). We 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 the 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 [12], 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:

- λ is the mean arrival rate of the requests/events.
- μ 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 ρ, which presents the utilization of processors. Then:

$$\rho = \frac{\lambda}{\mu N_a \epsilon(N_p)} \tag{1}$$

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 Π_W to be the probability that an event request has to wait. So Π_W is the sum of the probabilities this system contains i requests p_i, where $i \geqslant c$:

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

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

Substitute c with N_a, then:

$$\Pi_W = \frac{(N_a\rho)^{N_a}}{N_a!}\left((1-\rho)\sum_{n=0}^{N_a-1}\frac{(N_a\rho)^n}{n!} + \frac{(N_a\rho)^{N_a}}{N_a!}\right)^{-1} \tag{2}$$

Then the mean waiting queue length L_q is:

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

Then the mean queue length L is:

$$L = L_q + \frac{\lambda}{\mu\epsilon(N_p)} = \Pi_W \cdot \frac{\rho}{1-\rho} + \frac{\lambda}{\mu\epsilon(N_p)}$$

According to Little's Law [13], the average waiting time W_q is:

$$W_q = \frac{L_q}{\lambda} = \Pi_W \cdot \frac{1}{1-\rho} \cdot \frac{1}{\mu N_a \epsilon(N_p)}$$

Then the mean flow time of W in $M/M/N_a/N_p$ is [14]:

$$W_{M/M/N_a/N_p} = W_q + \frac{1}{\mu\epsilon(N_p)} = \Pi_W \cdot \frac{1}{1-\rho} \cdot \frac{1}{\mu N_a \epsilon(N_p)} + \frac{1}{\mu\epsilon(N_p)} \tag{3}$$

Mean flow time $(M/D/N_a/N_p)$**.** Then we calculate 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 [15]. Therefore, in order to get the mean flow time for $M/D/N_a/N_p$, we apply the following approximation proposed in [16]:

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

Apply Little's Law, and substitute c and ρ. We get the mean flow time in $(M/D/N_a/N_p)$:

$$W_{M/D/N_a/N_p} = L^{app}_{M/D/c} \cdot \frac{1}{\lambda}$$

$$= \frac{1}{2}\left[1 + (1 + \frac{\lambda}{\mu N_a \epsilon(N_p)})(N_a - 1)\frac{(\sqrt{4+5N_a}-2)\mu\epsilon(N_p)}{16\lambda}\right] \cdot W_{M/M/N_a/N_p} \tag{4}$$

2.4 | Discussion of the model

2.4.1 | Relationship between flow time and processor utility

According to equation 3, Figure 2 shows a plot of the relationship between W and ρ 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 ρ is above 80%. This leads to a dilemma that the system cannot reach both very high utilization and high performance, i.e that is 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 number of N_a and N_p since it causes a low processor utilization. In practice, keeping the utilization ρ between 70% to 80% is considered a good operational level, without degrading much performance.

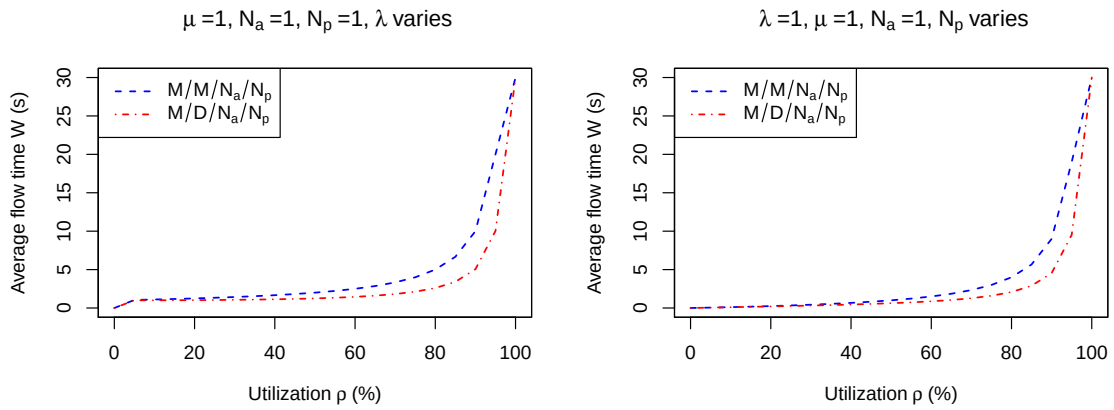


FIGURE 2 The theoretical relationship between processor utilization ρ and the mean event-handling flow time W in an event-driven system.

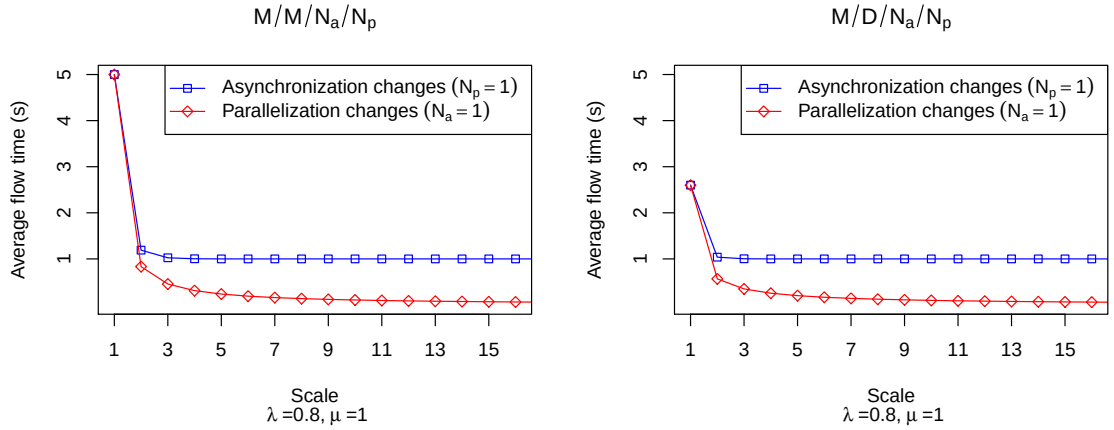


FIGURE 3 The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 0.8, \mu = 1$).

2.4.2 | 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 3 and Figure 4 show the results when the total number of processors is fixed as 16, 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 5 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 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 ρ, 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.

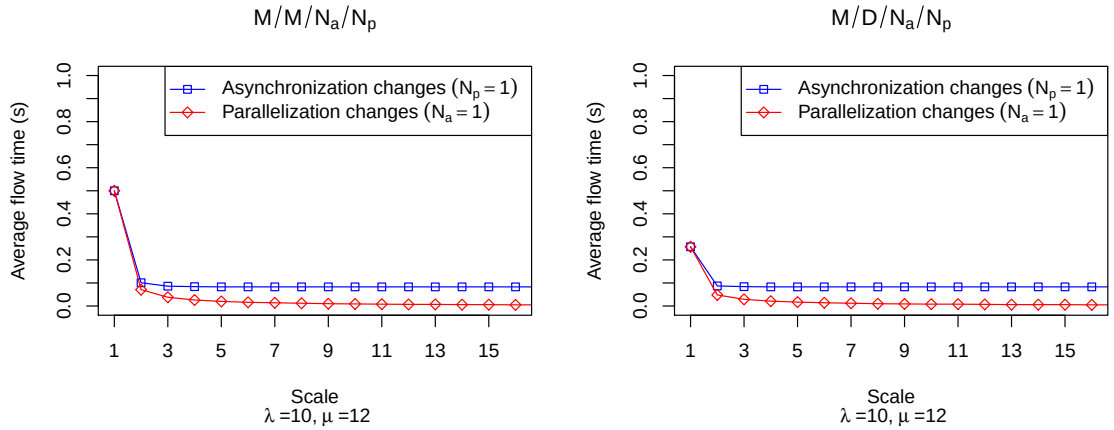


FIGURE 4 The performance comparison between merely increasing asynchronous scale or parallel scale ($\lambda = 10, \mu = 12$).

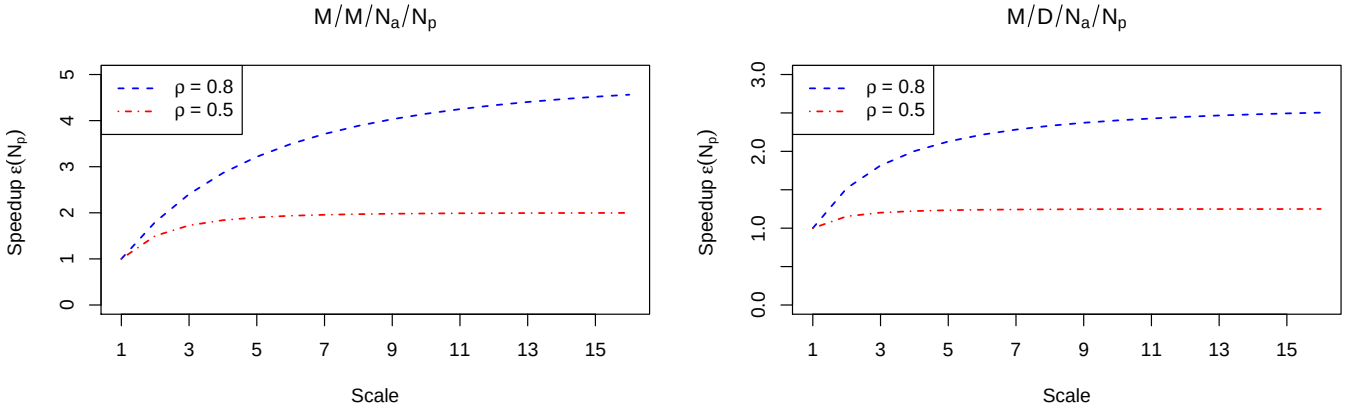


FIGURE 5 Minimum speedups $\epsilon(N_p)$ should be achieved that make the same W when the scale are applied to asynchronization.

3 | EVENT-DRIVEN PARALLELIZATION WITH OPENMP EXTENSION

After discussing the performance model for parallelized event-driven execution in the previous section, we now look at our proposed approach to support such an execution with OpenMP. The extension is based on the virtual target concept[6]. This approach is complementary to the standard OpenMP fork-join model and it enables code with a concurrent and asynchronous nature using OpenMP. By comparing this concept with the OpenMP task constructs, we analyze the advantages for parallelizing event-driven programs with the virtual target concept.

3.1 | Virtual Target

In order to offload computation from the current executing thread, the proposed syntax in Figure 6 is inspired by the `target` directive introduced in the OpenMP 4.0 specification. The initial purpose of the `target` directive is to utilize available accelerators in addition to multi-core processors in the system. The `target` directive offloads the computation of its code block to a specified accelerator, if a `device` clause is followed.

3.1.1 | Directive syntax extensions

Virtual target. In line with the existing `target` directive, this extension of the target syntax introduces the concept of a *virtual target*. A *virtual target* means the computation is not offloaded to a real physical device. Instead, it is a software-level executor capable of asynchronously offloading the target block from the thread which encounters this `target` directive. Conventionally, a *device target* has its own memory and data environment,

Name	`virtual_target_register()`	`virtual_target_create()`
Parameters	`tname`:String	`tname`:String, `n`:Integer
Description	The thread which invokes this function will be registered as a virtual target named `tname`.	Creating a worker virtual target with maximum of `n` threads, and its name is `tname`.

TABLE 1 Runtime functions to create virtual targets in Pyjama.

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* still actually shares the same memory that the host holds, so the data context remains the same when entering the target code block. Generally, a *virtual target* is a syntax-level abstraction of a thread pool executor, such that the `target` block is asynchronously executed by the executor specified by the *target-name*. Therefore, 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.

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. While this is defying the asynchronous nature of the virtual target, it corresponds to the expected standard behavior of other constructs in OpenMP. A more flexible and expressive control flow of the encountering thread can be achieved by adopting the *asynchronous-property-clause*. Using these clauses, a target block can be regarded as an asynchronous task. Figure 7 lists all the asynchronous properties a target code block can have in the program. `nowait` has the expected behavior that the target code block is bypassed by the encountering thread. The same is true for the `name_as/wait` option, but the encountering thread can query wait later for the completion of the target block. Lastly, `await` is like `wait` for the encountering thread, but during the wait other event handlers are executed. A more detailed explanation of *asynchronous-property-clause* can be found in [6].

Runtime Support. A virtual target is essentially a thread pool executor, or can be an event dispatching thread. Its lifecycle lasts throughout the program. Conceptually, a virtual target represents a type of execution environment defining its thread affiliation (to ensure operations that are not thread-safe are only executed by a specific thread), and scale (confine the number of threads of a thread pool) [this sentence is confusing, please rewrite]. This design enables programmers to flexibly submit different code target blocks to different execution environments. Table 1 describes the additional OpenMP APIs provided for managing virtual targets, and implemented in the Pyjama runtime.

3.2 | Comparing 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.

Listing 1 and Listing 2 show the two simple examples of using these approaches. 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 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 8). With OpenMP tasks, only one thread pool is managed to which all tasks are en-queued to a single queuing mechanism (whether it be work-sharing or work-stealing). In contrast, more than one tasking queue mechanism 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 4.2.

A summary of the differences between OpenMP tasks and virtual target is listed in Table 2 .

Listing 1: The example of using OpenMP tasks.

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

Listing 2: The example of using virtual targets.

```
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");
}
```

Distinction	OpenMP Tasking	Virtual Target
Objective	Task parallelization	Concurrency
Scenario used	Task decomposition and parallelization	Event-driven offloading and context switching
Effective region	Only in parallel region	Everywhere
Dependency handling	Data dependency	Control flow dependency
Task pool number	Single	Multiple

TABLE 2 Comparison between OpenMP tasking and target virtual.

4 | EVALUATION

This sections presents a new evaluation of the parallelized event-driven programming approach using the virtual target extension for OpenMP. By doing this, we reflect back on the performance model developed in Section **??**. In order to do this we undertake two experiments.The first experiment evaluates how the responsiveness of the EDT is improved in a GUI application using the proposed approach. The second experiment shows how the event handling flow times are reduced by using a customized combination of parallelization and asynchronization.

4.1 | Responsiveness of the EDT

To study the responsiveness of the EDT, an interactive application was developed in Java called ParaImage [17], which is a desktop application for image search and manipulation. This experiment is executed on a desktop system, which is equipped with Intel 3.40GHz i5-3570 CPU 4, and 8GB RAM.

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 [6]. When doing this with OpenMP-like directives, the parallelization is virtually free and the code logic remains sequential. To illustrate this, Listing 3 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

Listing 3: The search listener function implementation.

```
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));
      }
  }
}
```

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.

In order to profile the latency of the graphical user interface, an instrumentation tool LagHunter [8] is used to trace the call stack of the EDT in the runtime. The output of LagHunter is analyzed post-mortem with LagAlyzer [9], 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 that users can retrieve images from the Internet by inputting keywords 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 being 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 cancel button, remain enabled. Unlike the sequential, synchronous version, users are now able to cancel the remainder of the retrieval midway during the search.

Figure 9 shows the profiling visualization of the EDT in both the sequential and asynchronous versions. The horizontal axis shows time and the vertical axis symbolizes the call-stack. The visualization traces the method call from the EDT and methods are categorized in different colors according to their properties. Table 3 explains the details. In the sequential implementation, the dispatching clearly covers large parts of the lifetime of the EDT. A click to the search button leads to a synchronous processing and the EDT remains unresponsive until the event handling is completed. 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. In this search scenario, after applying the OpenMP directives, the EDT idle rate increases from 74.5% to 91.0% compared to 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 [18]. By applying this threshold as the criterion, a Cumulative Latency Distribution [19] is given in Figure 10 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 cause 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.

4.2 | Performance of parallelized event-handling

In this section we evaluate the performance of a parallelized event-handling application. To do so we implemented a benchmark 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 [20].

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 kernel. The parameter values of each kernel are listed in Table 4 .

We use the notation PxAy 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 $N_a = y_1 + y_2 + y_3 + y_4$.

4.2.1 | Adjust asynchronization to decrease queue time

In the first experiment, we do not use parallelism, but vary the asynchronization. We implemented the asynchronous versions in two different ways. First, we use traditional OpenMP task directive to offload event handler executions to the parallel region thread pool **OpenMP64**,then there are totally 64 asynchronous works are used **(P1A64)**. We can use this approach here as we can adjust the EDT in our synthetic application (which is not always possible, see discussion in Section 3.2). Second, we use virtual targets 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 11 compares the performance of the three approaches.

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. This implementation is achieved by using the traditional OpenMP tasking programming style (shown in Listing 1), in which the parallel region starts at the application startup, and a single thread is responsible for submitting tasks. 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 the high utilization, the number of asynchronous workers for each kernel handler is calculated by $N_a = \lceil \frac{\lambda}{\mu\rho} \rceil$ (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 conclude 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 **??**, a succinct use of processors can be achieved without significantly degrading the performance.

4.2.2 | Using both asynchronization and parallelization to decrease flow time

We now repeat the previous experiments, but also adjust 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 12 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, we see 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 clear inferior solutions.

Then we try to redistribute the processors for a better performance. Comparing 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 2.4.2, increasing parallelization scale can achieve better performance than applying asynchronization scale.

Figure 13 shows 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 is more worthwhile than increasing the asynchronization scale. As discussed in Section 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 13 indicates the kernel Series and RayTracer have better parallelism scale, allocating more processors to parallelelize 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 14 verifies this using the corresponding thread distribution.

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. Comparing to the previous distribution shown in Section 4.2.1, the results show that the new distribution has a better performance then the purely asynchronous distribution (left in Figure 14).

According to the speedups in Figure 13 , 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 14 shows another distribution where a total of 24 processors are used. Comparing 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.

5 | RELATED WORK

In this section we look at the directly and indirectly related work.

5.1 | Queue Theory

Queue theory is a mathematical model to study the processing of requests in queues. This research was originally started by Erlang and he created models to describe the Copenhagen telephone exchange system [21]. In order to formally describe the property of the queue, Kendall Notation [11] is widely used. Some basic models have been extensively researched such as $M/M/1$, $M/D/1$, $M/G/1$. As an extension, models with multiple servers such $M/M/k$ and $M/D/k$ also drew attention and have been investigated. Different key metrics are used to evaluate the performance of the queue. Some of the most important metrics are flow time, e.g. distribution / mean flow time, waiting time distribution / mean waiting time, and system utilization.

5.2 | Concurrency

Asynchronous programming is traditionally used in single-threaded applications to achieve cooperative multitasking [22]. 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 build-in or extended library interfaces to support asynchronous programming. For example, C++11 provides std::async, while Java provides the Future interface [23] building asynchronous computations. Java NIO libraries [24] provide non-blocking and asynchronous I/O operations. Microsoft .NET provides three types of asynchronous programming patterns [25]: (1) Asynchronous Programming Model (APM); (2) Event-based Asynchronous Pattern (EAP); (3) Task-based Asynchronous Pattern (TAP).

Languages support. Unlike libraries, language-level support for asynchronization tends to require less code restructuring. Fischer et al. [26] 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[27]. New language designs also tend to support asynchronization. For example, P[28] is a domain-specific language for the modeling of state machines, and all machines communicate via asynchronous events. Eve[29] is another parallel event-oriented language for the development of high-performance I/O applications. Other language-level concepts such as the actor model[30,31] and co-routines[32] provide variations to asynchronization.

5.3 | 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[33] and provides interfaces for task submissions. Some languages support tasks at a language level, such as Cilk[34] and JCilk[35]. In addition to the actual parallelization, handling task dependencies and code restructuring is another challenge faced. Parallel Task[36], as a language extension of Java, supports task creation and dependency handling. OoOJava[37] and DOJ[38] both introduce the task keyword to achieve out-of-order execution of the code blocks, with the support of automatic dependency analysis between tasks. Efforts also exist in introducing real-time guarantees to OpenMP[??].

Apple's Grand Central Dispatch (GCD)[?] is a tasking system that combines queues with closures called blocks. Programmers explicitly enqueue blocks into the main queue and into local and global queues they've created. The system schedules this enqueueing on the basis of the available cores' priorities. The two major differences between GCD and virtual target concepts are as follow:

1. Virtual targets enqueue code blocks using directives, which does not directly change the logic of the sequential code; In contrast, Apple provides GCD as a library and programmers have to explicitly call the functions to achieve the code offloading;
2. GCD is a system-level support which manages the sharing of system resources among different applications. As a comparison, the purpose of virtual target is used to facilitate the application and the application runtime is in charge of the executors, thread pools, event dispatching thread, etc.

6 | CONCLUSION

This paper 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 threads/processors should be distributed correspondingly if the event handlers have potential to be efficiently parallelized. 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. Future work involves an extension of the performance model to consider the architectural details such as hyper-threading, turbo-boost and I/O-heavy operations. The relationship between parallelization scale and asynchronization scale can also be further studied, exploring the solution to dynamically maximize the event-driven performance in a parallel system.

References

1. Ayguade E., Copty N., Duran A., et al. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems.* 2009;20(3):404-418.
2. Duran Alejandro, Perez Josep M., Ayguadé Eduard, Badia Rosa M., Labarta Jesus. Extending the OpenMP Tasking Model to Allow Dependent Tasks:111–122. Berlin, Heidelberg: Springer Berlin Heidelberg 2008.
3. Teruel Xavier, Klemm Michael, Li Kelvin, Martorell Xavier, Olivier Stephen L., Terboven Christian. A Proposal for Task-Generating Loops in OpenMP:1–14. Berlin, Heidelberg: Springer Berlin Heidelberg 2013.
4. Balart Jairo, Duran Alejandro, Gonzàlez Marc, Martorell Xavier, Ayguadé Eduard, Labarta Jesús. Experiences Parallelizing a Web Server with OpenMP:191–202. Berlin, Heidelberg: Springer Berlin Heidelberg 2008.
5. Salva Sébastien, Delamare Clément, Bastoul Cédric. Web Service Call Parallelization Using OpenMP:185–194. Berlin, Heidelberg: Springer Berlin Heidelberg 2008.

6. Fan X., Sinnen O., Giacaman N.. Towards an Event-Driven Programming Model for OpenMP. In: :240-249; 2016.

7. Vikas , Giacaman Nasser, Sinnen Oliver. Multiprocessing with GUI-awareness using OpenMP-like directives in Java. *Parallel Computing*. 2013;(accepted for publication).

8. Jovic Milan, Adamoli Andrea, Hauswirth Matthias. Catch Me if You Can: Performance Bug Detection in the Wild. In: OOPSLA '11:155–170ACM; 2011; New York, NY, USA.

9. Adamoli A., Jovic M., Hauswirth M.. LagAlyzer: A latency profile analysis and visualization tool. In: :13-22; 2010.

10. Fan Xing, Sinnen Oliver, Giacaman Nasser. Asynchronous OpenMP Tasking with Easy Thread Context Switching and Pool Control:217–230. Cham: Springer International Publishing 2017.

11. Kendall David G. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*. 1953;:338–354.

12. Haight Frank Avery. Handbook of the Poisson distribution. 1967;.

13. Little John DC. A proof for the queuing formula: L= λ W. *Operations research*. 1961;9(3):383–387.

14. Braband Jens. Waiting time distributions for M/M/N processor sharing queues. *Stochastic Models*. 1994;10(3):533–548.

15. Tijms Henk. New and old results for the M/D/c queue. *{AEU} - International Journal of Electronics and Communications*. 2006;60(2):125 - 130.

16. Cosmetatos George P.. Some Approximate Equilibrium Results for the Multi-Server Queue (M/G/r)*. *Journal of the Operational Research Society*. 1976;27(3):615–620.

17. Nicolau Peter. *Using Parallel Task for Parallelising an Object-Oriented Desktop Application*. 2013.

18. Jovic Milan, Hauswirth Matthias. Measuring the performance of interactive applications with listener latency profiling. In: :137–146ACM; 2008.

19. Jovic Milan, Adamoli Andrea, Zaparanuks Dmitrijs, Hauswirth Matthias. Automating Performance Testing of Interactive Java Applications. In: AST '10:8–15ACM; 2010; New York, NY, USA.

20. Bull J Mark, Smith Lorna A, Westhead Martin D, Henty David S, Davey Robert A. A benchmark suite for high performance Java. *Concurrency - Practice and Experience*. 2000;12(6):375–388.

21. Allen Arnold O. *Probability, statistics, and queueing theory*. Academic Press; 2014.

22. Engelschall Ralf S.. Portable Multithreading: The Signal Stack Trick for User-space Thread Creation. In: ATEC '00:20–20USENIX Association; 2000; Berkeley, CA, USA.

23. Oracle . Java 7 Future Interface Available at: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html2016.

24. Oracle . Java I/O, NIO, and NIO.2 Available at: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html2016.

25. Microsoft . Asynchronous Programming Patterns Available at: https://msdn.microsoft.com/en-us/library/jj152938%28v=vs.110%29.aspx2016.

26. Fischer Jeffrey, Majumdar Rupak, Millstein Todd. Tasks: language support for event-driven programming. In: :134–143ACM; 2007.

27. Okur Semih, Hartveld David L, Dig Danny, Deursen Arie van. A study and toolkit for asynchronous programming in C#. In: :1117–1127ACM; 2014.

28. Desai Ankush, Gupta Vivek, Jackson Ethan, Qadeer Shaz, Rajamani Sriram, Zufferey Damien. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*. 2013;48(6):321–332.

29. Fonseca Alcides, Rafael João, Cabral Bruno. Eve: A Parallel Event-Driven Programming Language. In: :170–181Springer; 2014.

30. Hewitt Carl, Bishop Peter, Steiger Richard. A universal modular actor formalism for artificial intelligence. In: :235–245Morgan Kaufmann Publishers Inc.; 1973.

31. Haller Philipp, Odersky Martin. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*. 2009;410(2):202–220.

32. De Moura Ana Lúcia, Rodriguez Noemi, Ierusalimschy Roberto. Coroutines in lua. *Journal of Universal Computer Science*. 2004;10(7):910–925.

33. Rudolph Larry, Slivkin-Allalouf Miriam, Upfal Eli. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In: SPAA '91:237–245ACM; 1991; New York, NY, USA.

34. Frigo Matteo, Leiserson Charles E., Randall Keith H.. The Implementation of the Cilk-5 Multithreaded Language. In: PLDI '98:212–223ACM; 1998; New York, NY, USA.

35. Danaher John S., Lee I.-Ting Angelina, Leiserson Charles E.. Programming with exceptions in JCilk. *Science of Computer Programming*. 2006;63(2):147 - 171. Special issue on synchronization and concurrency in object-oriented languages.

36. Giacaman N., Sinnen O.. Task Parallelism for Object Oriented Programs. In: :13-18; 2008.

37. Jenista James Christopher, Eom Yong hun, Demsky Brian Charles. OoOJava: Software Out-of-order Execution. In: PPoPP '11:57–68ACM; 2011; New York, NY, USA.

38. Yang Stephen, Jenista James C, Demsky Brian, others . DOJ: Dynamically parallelizing object-oriented programs. In: :85–96ACM; 2012.

//#omp target virtual(*name-tag***)** *[clause[[,]clause]...]*
structured-block

clause:

data-handling-clause

asynchronous-property-clause

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

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

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

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

FIGURE 6 Extended target directive.

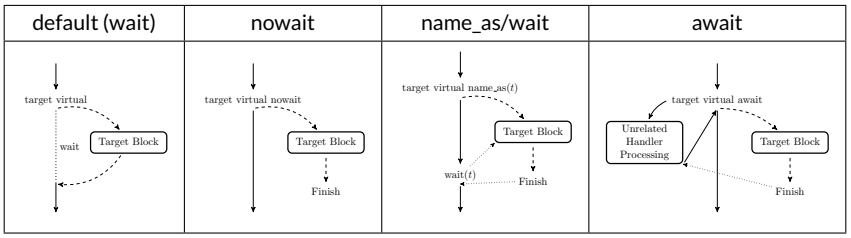


FIGURE 7 Different asynchronous modes, by using different *asynchronous-property-clauses*.

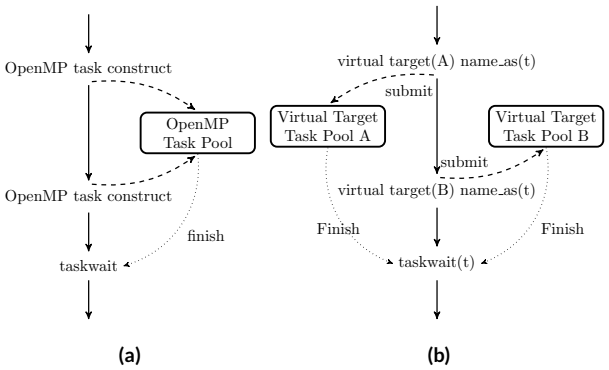
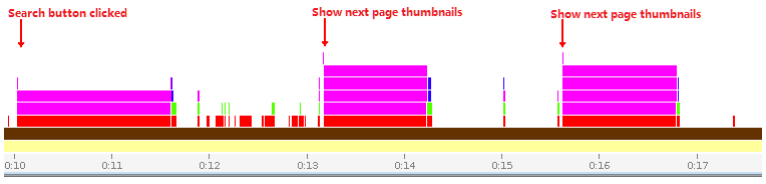
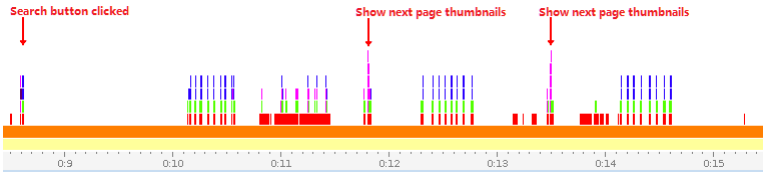


FIGURE 8 The demonstration of OpenMP tasking (a) and virtual target (b).



(a) EDT profiling in sequential version.



(b) EDT profiling in the asynchronous version, implemented by virtual target directives.

FIGURE 9 The EDT execution profiling on the searching scenario.

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

TABLE 3 Interval types and their represent colors.

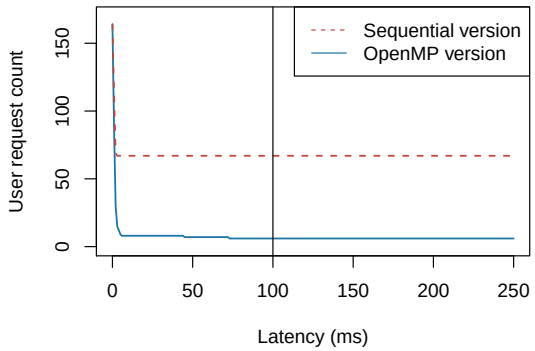


FIGURE 10 Cumulative Latency Distributions.

Computational kernel	T_{seq}	λ	$\mu(\frac{1}{T_{seq}})$
Crypt	177ms	10	5.65
MonteCarlo	496ms	4	2.02
Series	779ms	10	1.28
RayTracer	1175ms	4	0.85

TABLE 4 Computational kernels and their arrival rates and sequential service rates.

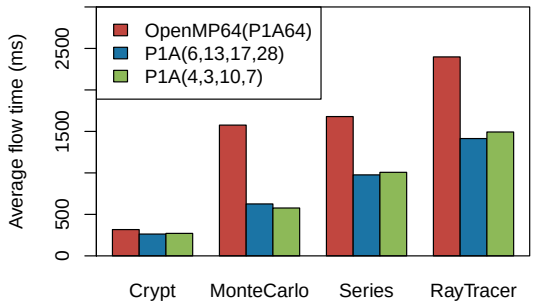


FIGURE 11 The mean flow times (ms) of four kernels implemented by three different asynchronization scale distributions.

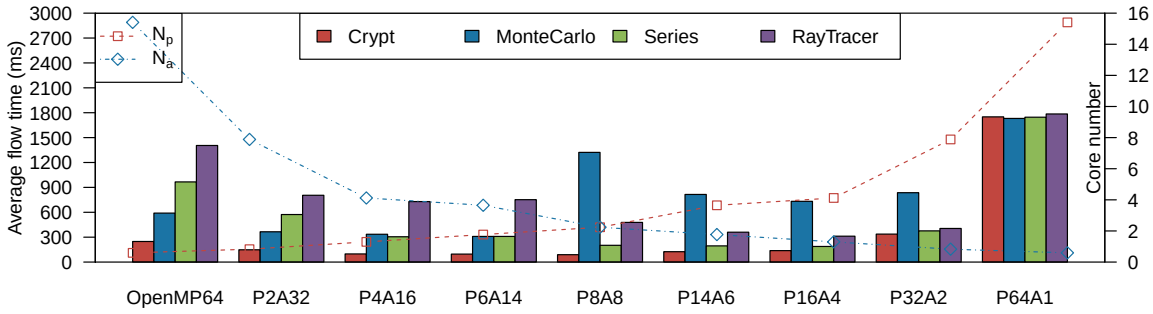


FIGURE 12 64-processor distribution and its mean flow time (ms), categorized by different kernel handlers.

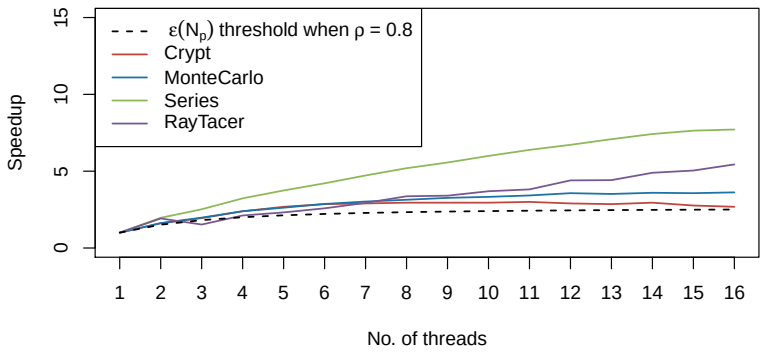


FIGURE 13 Parallel speedup of four kernels and the minimum speedup threshold for $\rho = 0.8$.

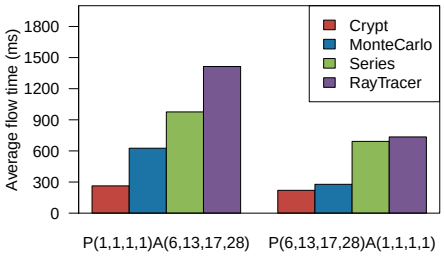


FIGURE 14 Performance comparison of two types of distribution between asynchronization and parallelization with total amount of 64 processors.

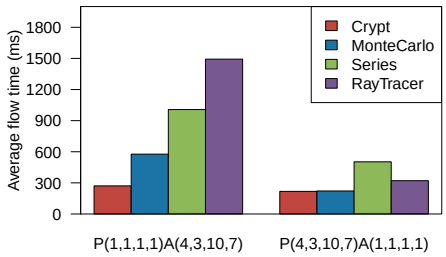


FIGURE 15 Performance comparison of two types of distribution between asynchronization and parallelization with total amount of 24 processors.