Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 318 (2015) 129–142

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Dynamic Subtask Dispersion Reduction in Heterogeneous Parallel Queueing Systems

Tommi Pesu and William J. Knottenbelt[1,2]

Department of Computing, Imperial College London, South Kensington Campus, SW7 2AZ

Abstract

Fork-join and split-merge queueing systems are mathematical abstractions of parallel task processing systems in which entering tasks are split into N subtasks which are served by a set of heterogeneous servers. The original task is considered completed once all the subtasks associated with it have been serviced. Performance of split-merge and fork-join systems are often quantified with respect to two metrics: task response time and subtask dispersion. Recent research effort has been focused on ways to reduce subtask dispersion, or the product of task response time and subtask dispersion, by applying delays to selected subtasks. Such delays may be pre-computed statically, or varied dynamically. Dynamic in our context refers to the ability to vary the delay applied to a subtask according to the state of the system, at any time before the service of that subtask has begun. We assume that subtasks in service cannot be preempted. A key dynamic optimisation that benefits both metrics of interest is to remove delays on any subtask with a sibling that has already completed service. This paper incorporates such a policy into existing methods for computing optimal subtask delays in split-merge and fork-join systems. In the context of two case studies, we show that doing so affects the optimal delays computed, and leads to improved subtask dispersion values when compared with existing techniques. Indeed, in some cases, it turns out to be beneficial to initially postpone the processing of non-bottleneck subtasks until the bottleneck subtask has completed service.

Keywords: dynamic dispersion reduction, fork-join, split merge, queueing networks

1 Introduction

Due to an ever increasing demand for performance and speed in the modern world and the eventual exhaustion of possible optimisations to single server systems, more and more of the world is turning towards parallel and distributed systems. This trend is especially apparent in the IT world where companies are adopting distributed storage facilities, multi-core processors, RAID systems [11,10] and huge distributed computing platforms. However, IT is not the only area where such demand is needed. For example, in finance equities are nowadays traded on a growing amount of distributed exchanges, manufacturers are making complex products

[1] Email: ttp09@imperial.ac.uk

[2] Email: wjk@doc.ic.ac.uk

http://dx.doi.org/10.1016/j.entcs.2015.10.023

with ever-growing distributed supply chains and in hospitals patient care involves a variety of parallel service stations [1].

Split-merge and fork-systems are parallel queueing network abstractions which describe task flow and processing in parallel networks. In such systems incoming tasks are split into a number of subtasks, each of which must be served before the whole task can be regarded as completed. Two primary performance metrics are of interest in such systems. *Task response time* is the time it takes from the point when a task enters the system to the point where all of the subtasks have been fully serviced. Response time has been a very intensively researched topic in queueing systems over the last 50 or so years, see e.g. [2,8,9]. *Subtask dispersion* is the difference in time between the completion of the first and last subtask. Subtask dispersion has not received much attention in the literature until recently, see e.g. [14,15,13]. Due to their synchronous nature, split-merge systems tend to have high task response time but low subtask dispersion. On the other hand, being asynchronous, fork-join systems tend to have low task response time but high subtask dispersion.

This paper examines split-merge and fork-join systems that use delays on the processing of subtasks to reduce subtask dispersion or the product of task response time and subtask dispersion. We make a distinction between two types of delay adjusting systems that in the past have not be clearly distinguished. In the first type of system once a delay is set, it cannot be changed. The processing of the subtask is begun only once the delay assigned to it has expired. We refer to these as *static delay* systems. In the second class of systems, it is possible to preemptively modify (or cancel) the delay of a subtask at any time before it has begun service. However, once service has begun it is not possible to interrupt the service. We refer to these as *dynamic delay* systems. We additionally assume the availability of instant notifications of events of interest, e.g. when any subtask finishes service.

In dynamic delay systems, it is beneficial in terms of both task response time and subtask dispersion to remove delays on any subtask that has a sibling that has already completed service. This paper defines a new way to calculate subtask delays in split-merge and fork-join systems that is able to incorporate this optimisation. We begin by exploring 2-server split-merge systems with deterministic and exponential service to offer some intuition behind our technique. We then proceed to a 3-server test case to demonstrate that our technique is able to deliver substantial reduction in subtask dispersion compared to existing methods.

2 Preliminaries

This section contains a brief introduction to terms that are fundamental to the understanding of this paper. The section includes an introduction to split-merge and fork-join systems. Both systems are queueing network models for describing the processing of a set of subtasks in parallel. In addition it describes related quantitative metrics, including response time, subtask dispersion and a trade-off metric. The trade-off metric can be used to make decisions when both subtask

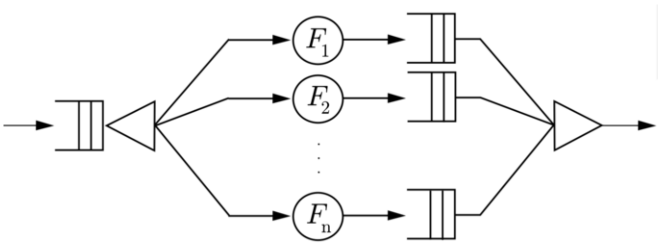

Fig. 1. A split-merge system [12].

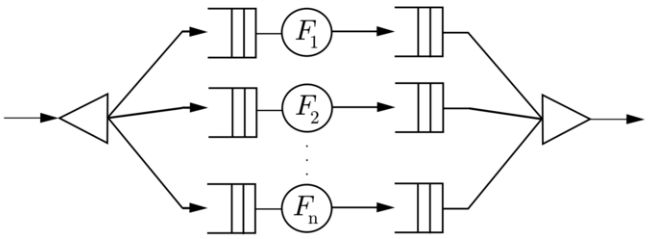

Fig. 2. A fork-join system [12].

dispersion and task response time are regarded as important.

2.1 Parallel Queueing Systems

2.1.1 Split-Merge

In the split-merge system considered in this paper arriving tasks have an interarrival rate that is exponentially distributed with a rate of λ. The system structure is shown in Figure 1. If no task is currently in service an arriving task enters service straight away. Otherwise it enters a queue to wait for its turn. When a task completes service it leaves the system.

When a task enters service it is split into N subtasks. Each of these subtasks is then processed by its own server. The service time of each server is characterised by a probability distribution. The task is considered to be done with service once all N subtasks have been serviced by their respective servers.

2.1.2 Fork-Join

As shown in Figure 2, the fork-join system is quite similar in structure to the split-merge system, but buffering of incoming tasks takes place at subtask-level instead of at task-level. The tasks again have an interarrival rate that is exponentially distributed with a rate of λ. Arriving tasks are immediately split into N subtasks. Each individual server serving the subtasks then has its own queue. The subtask servers independently process all subtasks waiting in their own queue. Once all the subtasks of a task have been serviced by their corresponding server the task is considered complete.

2.2 Performance Metrics

2.2.1 Task Response Time

Task Response time is the length of time it takes for a task to get processed. The clock is started when the task first enters the system. Once all the subtasks belonging to that task have been completely serviced the clock stops.

For split-merge systems it is possible to calculate task response time analytically. In particular, the split-merge system can be thought of as an M/G/1 queue where the probability distribution for task service time is defined by the probability distribution of last subtask's finishing time. The latter can be straightforwardly calculated using the theory of heterogeneous order statistics, as shown in Equation (3). From this distribution it is then possible to calculate the mean and variance of the distribution, which can then be used in the Pollaczek–Khinchine formula for computing mean response time in M/G/1 queues:

$$E[R_\lambda(t)] = \frac{\rho + \mu\lambda Var[X_{(N)}]}{2(\mu - \lambda)} + \mu^{-1} \tag{1}$$

Here $\mu = E[X_{(N)}]$ is the service rate, λ is the task arrival rate and $\rho = \lambda/\mu$ is the utilization of the server.

For fork-join systems there currently exists no analytical formula to calculate task response time except for simple cases [4,5]. Simulation does, however, provide a route to approximating task response time with an arbitrary degree of accuracy.

2.2.2 Subtask Dispersion

For a given task, subtask dispersion is the difference between the finishing times of its first and last subtasks. If N subtasks begin service simultaneously, the expected finishing times of the first and last subtask can be calculated by using the theory of heterogeneous order statistics [3]. The cumulative distribution function for the time of the first subtask to finish is given by Equation (2) and for the time of the last by Equation (3).

$$F_1(t) = Pr\{X_{(1)} < t\} = 1 - \prod_{i=1}^{N}[1 - F_i(t)] \tag{2}$$

$$F_N(t) = Pr\{X_{(N)} < t\} = \prod_{i=1}^{N}[F_i(t)] \tag{3}$$

Heterogeneous order statistics be used to define expected subtask dispersion $E[D]$ in the following way, which is shown in Equation (4) and (5).

$$E[D] = \int_0^\infty F_1(t) - F_N(t)\mathrm{d}t \tag{4}$$

$$E[D] = \int_0^\infty 1 - \prod_{i=1}^{N}(1 - F_i(t)) - \prod_{i=1}^{N} F_i(t)\mathrm{d}t \tag{5}$$

The way subtask dispersion is calculated here has been used to calculate subtask dispersion of split-merge systems [15,13] and for analysing instantaneous configurations of fork-join systems [14]. In the case of the fork-join algorithm in [14] subtasks are set to start processing immediately after a sibling task finishes if they have not started already. However, Equation 5 does not take this into account and is therefore unable to minimise the dispersion of a dynamic parallel system correctly. In Section 3.1 a new formula is derived that is able to do this.

2.2.3 *Trade-Off Metric*

Sometimes both subtask dispersion and task response time are important. In these cases it is possible to measure the effectiveness of the system with a trade-off metric defined as the product of subtask dispersion and task response time [15], i.e.

$$T(\lambda, t) = E[D(t)]E[R_\lambda(t)] \tag{6}$$

A similar metric has been explored in the context of the energy–response time product analysis of power policies for server farms [6,7]. For split-merge systems the trade-off equation can be expressed as:

$$T(\lambda, t) = \left[\int_0^\infty 1 - \prod_{i=1}^{N}(1 - F_i(t)) - \prod_{i=1}^{N} F_i(t)\mathrm{d}t\right]\left[\frac{\rho + \mu\lambda Var[X_{(n)}]}{2(\mu - \lambda)} + \mu^{-1}\right] \tag{7}$$

For fork-join systems the trade-off metric has to be quantitatively estimated via simulation , since – to the best of our knowledge – there are no closed form solutions for either subtask dispersion or task response time.

3 Method

This section introduces a way to calculate subtask dispersion in split-merge systems where a *start work* signal is sent to sibling subtasks once service of a subtask is completed. The model assumes that a delay applied to a subtask can be arbitrarily preempted at any point before service of the subtask has begun. However, once a subtask has begun service it will be serviced uninterrupted until it completes service. The *start work* signal is sent because removing delays after the first sibling subtask finishes service reduces both subtask dispersion and task response time.

3.1 *Calculating subtask dispersion in dynamic split-merge system*

It is explained here how the minimum subtask dispersion of a dynamic split-merge system can be obtained by choosing an appropriate delay vector $\mathbf{d}$. Equation (8) displays the formula that is minimised. The formula is split into two parts: $T(i, t, \mathbf{d})$ and $E_r(i, t, \mathbf{d})$.

The first function $T(i, t, \mathbf{d})$ calculates the probability that server i is the first server that finishes servicing its subtask at the time t, with the given delays $\mathbf{d}$. This result is calculated by multiplying the probability density function of server i

finishing at time t with the probability of all the other servers not finishing before time t. The mathematical formulation of this can be seen in Equation (9).

The second function $E_r(i, t, \mathbf{d})$ calculates how long the rest of the servers will take to complete their service, with the given delays $\mathbf{d}$. The average time for the rest of the servers to complete service is computed with the help of heterogeneous order statistics presented in [15].

$G_j(t, t', d_j)$ represents the probability distribution function of the service time of the jth server. If $t' < d_j$ then the jth server has not yet begun service of its subtask and servicing is begun immediately. Otherwise the server has already started servicing its subtask. In this case the service time is renormalised to take into account that some service has already been performed and the service of the subtask has not been completed. The two part G-function is the key difference compared to previous work presented in [14,15,13].

That is, we seek:

$$d_{\min} = \arg\min_{\mathbf{d}} \sum_{i=1}^{N} \int_0^\infty T(i, t, \mathbf{d}) E_r(i, t, \mathbf{d}) dt \tag{8}$$

where

$$T(i, t, \mathbf{d}) = f_i(t - d_i) \prod_{j \neq i} [1 - F_j(t - d_j)] \tag{9}$$

and

$$E_r(i, t', \mathbf{d}) = \int_0^\infty [1 - \prod_{j \neq i} G_j(t, t', d_j)] dt \tag{10}$$

with

$$G_j(t, t', d_j) = \begin{cases} F_j(t) & \text{if } t' < d_j \\ F_j(t - (t' - d_j) | t > 0) & \text{otherwise} \end{cases} \tag{11}$$

subject to conditions

$$\prod_{i=1}^{N} d_i = 0 \tag{12}$$

and

$$\forall i \quad d_i \geq 0 \tag{13}$$

The two conditions on $\mathbf{d}$ guarantee that no unnecessary delays are added and that delays are non-negative.

3.2 Deterministic 2-server dynamic split-merge example

In this section we will apply the subtask dispersion reduction formula above to a split-merge system with 2 servers. The service time densities of the servers are:

$$X_1 \sim \text{Det}(1)$$
$$X_2 \sim \text{Det}(2)$$

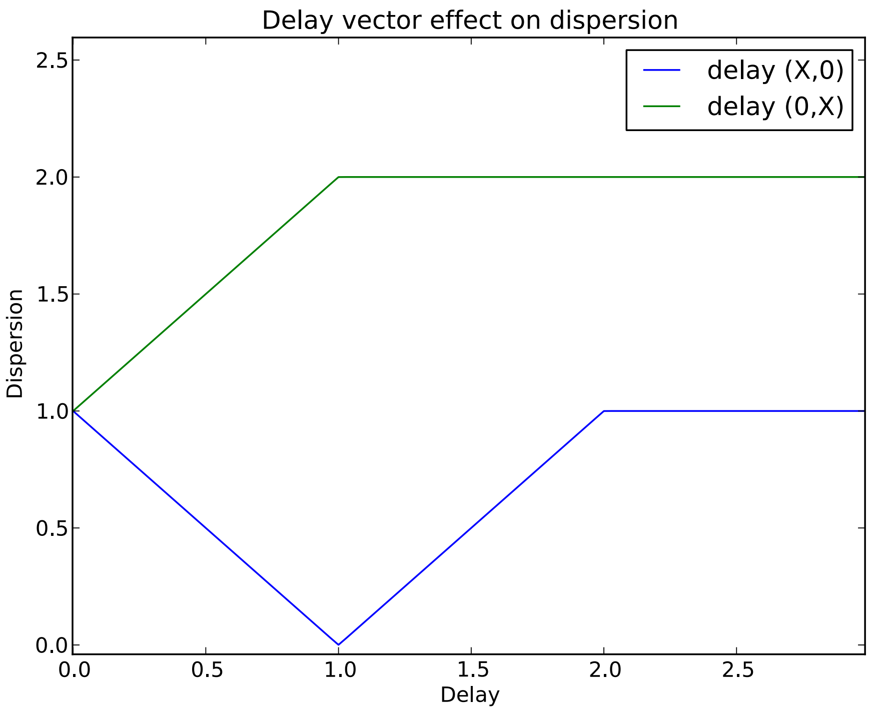

Fig. 3. Demonstration of how the delays affect dispersion in the two server deterministic example.

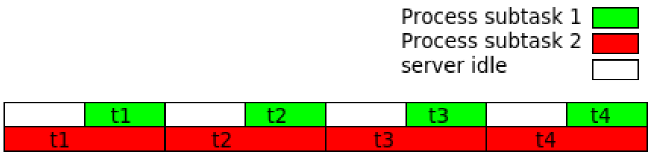

Fig. 4. Demonstration of how the deterministic two server case processes its tasks under heavy load.

Due to the deterministic function causing problems with integration our example will use uniform functions with a small range to approximate it as:

$$\mathrm{Det}(n) \approx \mathrm{Uni}(n - 0.001, n + 0.001) \tag{14}$$

The effect of varying $\mathbf{d}$ on subtask dispersion can be seen in Figure 3. The optimal delay is naturally $\mathbf{d} = (1, 0)$ and when the delays deviate from the optimal solution, subtask dispersion grows. Increasing server 1 delay past the optimal delay causes an increase that caps at 1. The capping is due to the dynamic delay interruption. Server 2 behaves similarly with the delay capped at 2. Figure 4 demonstrates how the system works with the optimal delay under a heavy load.

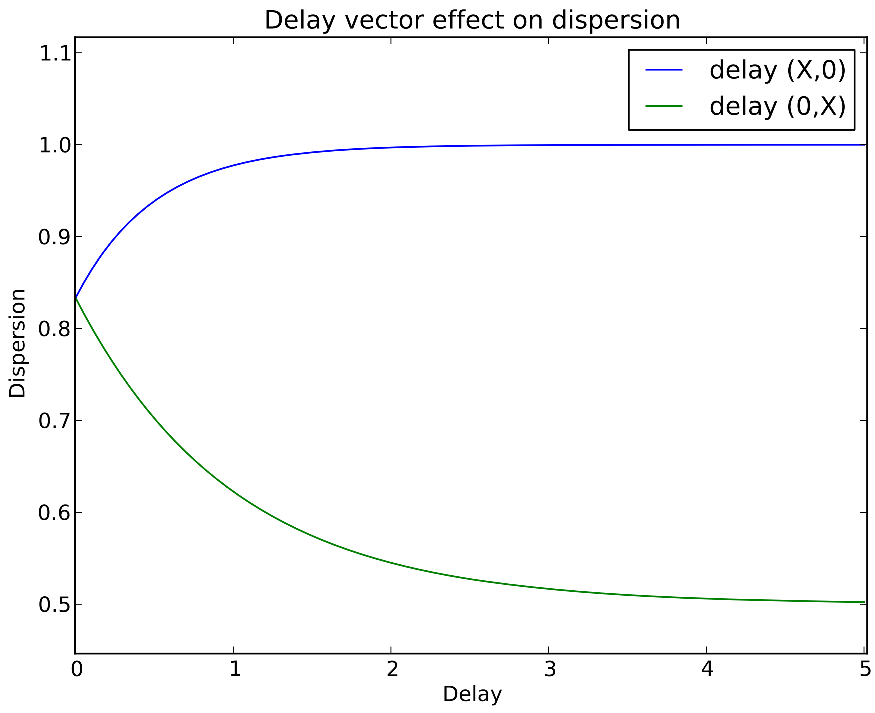

Fig. 5. Demonstration of how the delays affect subtask dispersion in the two-server exponential example.

3.3 Exponential dynamic 2-server split-merge example

In this section a split-merge system with exponentially distributed subtask service times is analysed. The exponential distribution has a parameter λ which is the inverse of average service time. The service time densities of the servers are:

$$X_1 \sim \text{Exp}(\lambda = 1)$$
$$X_2 \sim \text{Exp}(\lambda = 2)$$

Therefore the average service times of the two servers are 1 and 0.5 respectively.

The effect of varying $\mathbf{d}$ on subtask dispersion is shown in Figure 5. Intuitively, when a large delay is set on server 1, the subtask dispersion should increase towards 1.0. This is, because then the probability of server 2 finishing first increases towards 1. A similar argument can be done for setting a large delay on server 2 instead. As the delay grows towards infinity the chance of server 1 completing service first grows. The average service time of server 2 is 0.5. Therefore dispersion with infinite delay on server 2 is 0.5. The optimal delay that minimises dispersion is $\mathbf{d} = (0, \infty)$, as infinite delay on the server 2 guarantees that the server 1 will always finish first. Figure 6 demonstrates how the delays are applied in the two server exponential case. It can be seen that server 1 completes its service before service on server 2 is begun.

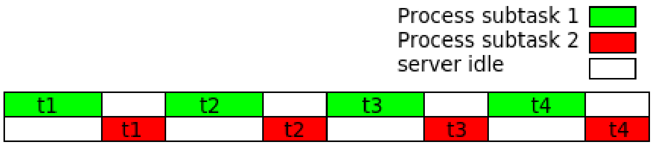

Fig. 6. Demonstration of how the exponential two-server case processes its tasks.

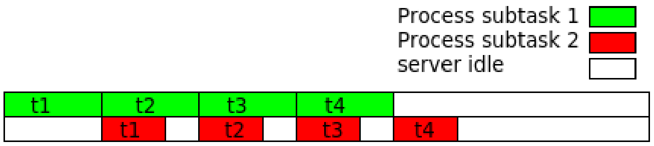

Fig. 7. An example of how idling time can be squeezed.

3.4 *Fork-join systems*

The throughput of a parallel system is maximised when at least one of the servers is performing work on a subtask throughout the time a task is being processed. If this is not the case as is in the exponential case in Figure 6, it is possible to decrease response time by removing idle time from the processing of each subtask. This is possible if using the sort of asynchronous task scheduled found in fork-join systems. This will, however, increase subtask dispersion. The effect of removing idling time on subtasks servicing can be observed in Figure 7.

4 Results

We present results of existing methods for subtask dispersion reduction in split-merge and fork-join systems and compare them against the algorithms described in this paper. The first case study uses the example from the paper [14] and the second uses a different configuration. The metrics of split-merge Methods 1, 2 and 3 are evaluated analytically. The metrics of fork-join Methods 4, 5 and 7 and split-merge Method 6 are simulated with 5 replicas of 5 million tasks each. The average task response time, subtask dispersion and trade-off metric were then calculated.

4.1 *Case Study 1*

The first test case sets the interarrival time of new tasks entering the system to be exponentially distributed with $\lambda = 0.78$ tasks per time unit. The service time densities of the parallel servers are:

$$X_1 \sim \text{Exp}(\lambda = 1)$$
$$X_2 \sim \text{Exp}(\lambda = 5)$$
$$X_3 \sim \text{Exp}(\lambda = 10)$$

The first five methods below are the same as in [14]. Methods 1–4 do not use interrupt to begin service immediately after a sibling task has finished (i.e. they are static delay methods) while Method 5 uses it (i.e. it is a dynamic delay method).

Method 1 represents a vanilla split-merge system where no delays are applied. The tasks are processed one at a time. The service of the next task does not begin before all the previous task's subtasks have finished. Corresponding performance metrics are:

Task response time:	5.195 time units
Subtask dispersion:	0.976 time units
Trade-off:	5.069 (time units)^2

Method 2 [15] represents a split-merge system where dispersion is minimised according to the formula described in Section 2.2.2. The resulting delays for subtasks are: $\mathbf{d} = (0, 0.524, 0.585)$. Corresponding performance metrics are:

Task response time:	33.638 time units
Subtask dispersion:	0.783 time units
Trade-off:	26.345 (time units)^2

Method 3 [13] represents a split-merge system where trade-off is minimised according to the formula described in Section 2.2.3. The resulting delays for subtasks are: $\mathbf{d} = (0, 0, 0.068)$. Corresponding performance metrics are:

Task response time:	5.286 time units
Subtask dispersion:	0.946 time units
Trade-off:	4.999 (time units)^2

Method 4 represents a vanilla fork-join system with no delays applied between subtasks. Each task is split into 3 subtasks which then each individually queue for their respective servers. Corresponding performance metrics are:

Task response time:	4.555 time units
Subtask dispersion:	4.480 time units
Trade-off:	20.406 (time units)^2

Method 5 represents a fork-join system with a dynamic subtask reduction algorithm [14]. This algorithm uses interruptions to start processing of sibling subtasks once a subtask finishes. The system uses definition of dispersion from Section 2.2.2. Corresponding performance metrics are:

Task response time:	4.675 time units
Subtask dispersion:	0.768 time units
Trade-off:	3.590 (time units)^2

The methods described next are described in Section 3 of this paper. They use interruptions to start processing of sibling tasks once a subtask finishes. The results have been calculated with the same simulation framework as the fork-join systems in the previous subsection.

Method 6 represents a split-merge system that uses the new dispersion calculation in Section 3.1 to calculate delays. The resulting initial delays for subtasks are: $\mathbf{d} = (0, \infty, \infty)$. Once the first subtask has completed the two remaining subtasks begin service immediately. Corresponding performance metrics are:

Task response time:	28.228 time units
Subtask dispersion:	0.233 time units
Trade-off:	6.581 (time units)^2

Method 7 modifies the split-merge system found in Method 6 to derive a fork-join system in which idling time is squeezed according to the principles of Section 3.4. Corresponding performance metrics are:

Task response time:	4.818 time units
Subtask dispersion:	0.269 time units
Trade-off:	1.296 (time units)^2

4.2 Case Study 2

The methods used in this case study are the same as the methods used above. The interarrival time of new tasks entering the system is exponentially distributed with $\lambda = 0.4$ tasks per time unit. The service time densities of the parallel servers are:

$$X_1 \sim \text{Exp}(\lambda = 1)$$
$$X_2 \sim \text{Exp}(\lambda = 2)$$
$$X_3 \sim \text{Exp}(\lambda = 2)$$

Method 1 represents a vanilla split-merge system where no delays are applied. Corresponding performance metrics are:

Task response time:	2.315 time units
Subtask dispersion:	1.083 time units
Trade-off:	2.508 (time units)^2

Method 2 [15] represents a split-merge system where dispersion is minimised according to the formula described in Section 2.2.2. The resulting delays for subtasks

are: $\mathbf{d} = (0, 0.288, 0.288)$. Corresponding performance metrics are:

Task response time:	2.777 time units
Subtask dispersion:	1.038 time units
Trade-off:	2.882 (time units)^2

Method 3 [13] represents a split-merge system where trade-off is minimised according to the formula described in Section 2.2.3. The resulting delays for subtasks are: $\mathbf{d} = (0, 0, 0)$. Corresponding performance metrics are:

Task response time:	2.315 time units
Subtask dispersion:	1.083 time units
Trade-off:	2.508 (time units)^2

Method 4 represents a vanilla fork-join system with no delays applied between subtasks. Each task is split into 3 subtasks which then each individually queue for their respective servers. Corresponding performance metrics are:

Task response time:	1.913 time units
Subtask dispersion:	1.627 time units
Trade-off:	3.114 (time units)^2

Method 5 represents a fork-join system with a dynamic subtask reduction algorithm [14]. This algorithm uses interruptions to start processing of sibling subtasks once a subtask finishes. The system uses definition of dispersion from Section 2.2.2. Corresponding performance metrics are:

Task response time:	2.227 time units
Subtask dispersion:	1.099 time units
Trade-off:	3.114 (time units)^2

Method 6 represents a split-merge system that uses the new dispersion calculation in Section 3.1 to calculate delays. This algorithm uses interruptions to start processing of sibling tasks once a task finishes. The resulting initial delays for subtasks are: $\mathbf{d} = (0, \infty, \infty)$. Once the first subtask has completed the two remaining subtasks begin service immediately. Corresponding performance metrics are:

Task response time:	4.671 time units
Subtask dispersion:	0.750 time units
Trade-off:	3.502 (time units)^2

Method 7 uses the split-merge system found in Method 6 to derive a fork-join system in which idling time is squeezed according to the principles of Section 3.4. Corresponding performance metrics are:

Task response time:	2.586 time units
Subtask dispersion:	0.921 time units
Trade-off:	2.381 (time units)^2

5 Conclusions

Two main conclusions can be drawn from the results. First the *start work* interrupt that removes delays on other subtasks preemptively once the first sibling subtask finishes is able to reduce subtask dispersion greatly. This can be seen when the subtask dispersion of Methods 1–5 and Method 6 and Method 7 are compared.

The improvement in subtask dispersion, however, comes at a cost to task response time. Method 6 is better in terms of subtask dispersion compared to the old dispersion minimisation technique Method 2, but it has a significantly higher response time when compared to the trade-off technique (Method 3) and vanilla technique (Method 1). However some of these issues can be corrected by Method 7 which is the fork-join version of Method 6. Method 7 does have a slightly increased subtask dispersion when compared to Method 6. However, task response time is still higher than found in the two other fork-join systems (Methods 4 and 5).

Secondly, the traditional method for calculating expected subtask dispersion is not appropriate for systems where subtask delay preemption is applied. In the first case study, the new method was able to reduce subtask dispersion by a factor of 3, which can be considered a major improvement.

In terms of future work, for both split-merge and fork-join systems, we aim to optimise for the trade-off metric and also investigate ways to support general service time distributions.

References

[1] S. W. M. Au-Yeung, P. G. Harrison, and W. J. Knottenbelt. Approximate queueing network analysis of patient treatment times. In *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, pages 45:1–45:12, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[2] François Baccelli, Armand M. Makowski, and Adam Shwartz. The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds. *Advances in Applied Probability*, 21(3):pp. 629–660, 1989.

[3] Herbert A. David and H. N. Nagaraja. *Wiley Series in Probability and Statistics.* John Wiley & Sons, 1980.

[4] L. Flatto and S. Hahn. Erratum: Two parallel queues created by arrivals with two demands I. *SIAM Journal on Applied Mathematics*, 45(1):168–168, 1985.

[5] Leopold Flatto. Two parallel queues created by arrivals with two demands II. *SIAM Journal on Applied Mathematics*, 45(5):pp. 861–878, 1985.

[6] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A. Kozuch. Optimality analysis of energy-performance trade-off for server farm management. *Performance Evaluation*, 67(11):1155 – 1171, 2010.

[7] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Server farms with setup costs. *Performance Evaluation*, 67(11):1123 – 1138, 2010.

[8] Peter Harrison and Soraya Zertal. Queueing models of {RAID} systems with maxima of waiting times. *Performance Evaluation*, 64(78):664–689, 2007.

[9] P. Heidelberger and K.S. Trivedi. Analytic queueing models for programs with internal concurrency. *IEEE Transactions on Computers*, C-32(1):73–82, Jan 1983.

[10] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Modelling Zoned RAID Systems using Fork-Join Queueing Simulation. In *6th European Performance Engineering Workshop (EPEW 2009)*, volume 5652 of *Lecture Notes in Computer Science*, pages 16–29, July 2009.

[11] Abigail S. Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Analytical and simulation modelling of zoned raid systems. *Comput. J.*, 54(5):691–707, May 2011.

[12] Iryna Tsimashenka. *Reducing Subtask Dispersion in Parallel Queueing Systems.* PhD thesis, Imperial College London, 2014.

[13] Iryna Tsimashenka, William Knottenbelt, and Peter Harrison. Controlling variability in split-merge systems. In Khalid Al-Begain, Dieter Fiems, and Jean-Marc Vincent, editors, *Analytical and Stochastic Modeling Techniques and Applications*, volume 7314 of *Lecture Notes in Computer Science*, pages 165–177. Springer Berlin Heidelberg, 2012.

[14] Iryna Tsimashenka and William J. Knottenbelt. Reduction of subtask dispersion in fork-join systems. In Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 325–336. Springer Berlin Heidelberg, 2013.

[15] Iryna Tsimashenka and William J. Knottenbelt. Trading off subtask dispersion and response time in split-merge systems. In Alexander Dudin and Koen De Turck, editors, *Analytical and Stochastic Modeling Techniques and Applications*, volume 7984 of *Lecture Notes in Computer Science*, pages 431–442. Springer Berlin Heidelberg, 2013.