Understanding the Effect of Task Granularity on Execution Time in Asynchronous Many-Task Runtime Systems

Shahzrad Shirzad[1](✉), R. Tohid[1], Alireza Kheirkhahan[1], Bibek Wagle[2], and Hartmut Kaiser[1]

[1] Louisiana State University, Baton Rouge, LA, USA
{sshirz1,akheir1}@lsu.edu, hkaiser@cct.lsu.edu
[2] The STE||AR Group, Baton Rouge, USA
bibek@alumni.lsu.edu

Abstract. Task granularity is a key factor in determining the performance of asynchronous many-task (AMT) runtime systems. The overhead of scheduling an excessive number of tasks with smaller granularities causes performance degradation, while creating a few larger tasks leads to starvation and therefore under-utilization of resources. In this paper, we developed an analytical model of the execution time of an application with balanced parallel *for-loops* in terms of grain size, and number of cores. The parameters of this model mostly depend on the runtime and the architecture. We introduce an approach to suggest a range of possible grain sizes to achieve the best performance based on the proposed model. To the best of our knowledge, our analytical model is the first to explain the relationship between the execution time in terms of grain size, runtime, and physical characteristics of the machine in an asynchronous runtime system.

Keywords: Task granularity · Analytical model · AMTs · HPX

1 Introduction

Achieving exascale computing relies on computing environments with complex architectures, deeper memory hierarchies, heterogeneous hardware and complex networks [7]. Asynchronous many-task (AMT) models and their corresponding runtimes are the solution to keep application developers safe from the upcoming architectures by mitigating exascale difficulties to runtime level [3]. New runtime systems rely on lightweight threads to avoid expensive context switching. Therefore, the cost of thread creation is relatively low, e.g., HPX threads are created in a few cycles. However, if millions of lightweight threads are created so each carry out a small task of a few cycles, then the overhead of task creation will be significant. On the other hand, if only a few tasks carry out the entire execution, resources will most likely not be utilized to the full extent. Therefore, the amount of work assigned to each task, *grain size*, requires meticulous analyses.

We have developed a model by carefully studying the relationship between the total execution time and the grain size. Based on the analytical model, we recommend a range of grain sizes that would lead us to lowest possible execution times. The model depends on two mostly architecture-specific parameters. Identifying these parameters on one system would then help us to improve the performance of other similar balanced for-loop applications on the same system.

The contribution of this work includes:

– Developing an analytical model to predict the total execution time of a balanced parallel for-loop. To our knowledge, this is the first analytical model in terms of both grain size and number of cores.
– A method has been offered to estimate the range of grain sizes to achieve minimum execution time for a particular number of cores.
– Building a microbenchmark on top of HPX to evaluate the model. The data collected through this microbenchmark is used to estimate model parameters on each machine architecture. The obtained parameters could then be utilized to predict the optimum range of grain sizes for minimum execution time, and consequently improve the performance of any other balanced parallel for-loop application executed on the same system.

2 Background

This section provides a brief overview of the concepts and technologies that are building blocks of this work.

2.1 HPX

HPX[11] is a C++ runtime system for parallel and distributed applications. HPX provides users with lightweight user-level threads with fast context switching [12]. When a thread is blocked, the scheduler picks up another one from the ready queue in order to hide latency, avoid starvation and therefore improve the utilization of the computation resources [12].

2.1.1 Execution Model

HPX's execution model mainly holds four factors responsible for performance degradation in parallel applications: Starvation, Latency, Overheads, and Waiting (also known as contention) [10].

Starvation refers to the situation where there is not enough work to keep the computing resources busy- this could be due to an insufficient total amount of work available, or unbalanced distribution of work among resources [12].

Latency is the time distance, usually measured in processor clock, between requesting and accessing remote data or services [4].

Overhead refers to the effort taken to manage parallel resources and actions on the critical path but are not necessarily needed by the application itself [12].

Waiting is the contention over shared physical or logical resources when one or more threads try to access the same resource and all get blocked [4].

Table 1. Table of notations

Parameter	Definition	Parameter	Definition
N	Number of cores	t_{seq}	Sequential execution time
n_t	Total number of tasks created	t_i	Execute time of one iteration
p_s	Total amount of work	n	Number of loop iterations
M	Number of utilized cores	c_s	Chunk size
t_{max}	Execution time of w_{max}	g	Grain size
w_{max}	Maximum amount of work assigned to a single core		

2.2 Analytical Modeling of Parallel Programs

The performance of a parallel program mostly depends on its underlying algorithm and the architecture it is run on [5]. Amdahl's law [2], Equation (1), shows that there is a limit on maximum speed-up achievable in a parallel application. This limit is imposed by the sequential fraction of the program denoted by σ.

$$S(p) = \frac{p}{1 + \sigma(p-1)} \tag{1}$$

Gunther [9] extends Amdahl's law by incorporating effects of three factors: *concurrency*, *contention*, and *coherency*, as shown in Eq. 2.

$$S(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)} \tag{2}$$

In this equation, known as the Universal Scalability Law (USL), concurrency (p) represents the linear speed-up in the absence of interactions among parallel processors, contention (σ) represents the serialization effect of shared writable data, and finally coherency or data consistency (κ) represents the cost imposed for keeping shared writable data consistent [9].

Several models have been proposed to model scalability, including the Geometric model [9], and the Quadratic model [8]. These models are mainly non-physical, and not applicable for large number of processors [8].

3 Methodology

In this section we provide an overview of the methodology used to analyze the effect of *grain size*, i.e., the workload of each runtime thread, on the execution time in AMT runtime systems. Table 1 shows the notations used throughout this section. It should be noted that the amount of work could be measured in terms of execution time or floating point operations depending on the application. Chunk size is defined as the number of iterations included in one task, while grain size is the amount of work contained in a task and is executed by a single user-level thread.

The total execution time is then defined as the maximum of the execution times of each individual core. In general this is the amount of time it takes for the core with maximum amount of work to finish execution of its task. Here the maximum expected amount of work to be assigned to a core is denoted as w_{max}, and the time it takes to execute this amount of work as t_{max}.

With this assumption, the key factors contributing to the execution time are, the overhead of scheduling tasks on the core with maximum amount of work, the time it takes to run w_{max} amount of work, denoted with t_{max}, and the number of cores executing the work(M). Depending on the amount of work available, either all N cores or less than N cores will be performing the work.

Equation (3) shows the expected formula in its simplest form.

$$execution\_time = t_{overhead} \;+\; t_{max} \tag{3}$$

$t_{overhead}$ represents the penalty that has to be paid for running the program in parallel. We hold two major factors accountable for this overhead.

The first factor is the overhead of scheduling the tasks. Although this overhead is negligible for a small number of tasks, it becomes significant as the number of created tasks becomes larger.

In the ideal case, when n_t tasks are created, $\lceil \frac{n_t}{N} \rceil$ of them would be scheduled on the core with the maximum amount of work. If we represent the overhead of scheduling one task on a core with α, then $\alpha\lceil \frac{n_t}{N} \rceil$ would be the scheduling overhead associated with $\lceil \frac{n_t}{N} \rceil$ tasks.

The second factor is the overhead due to contention and coherency based on USL. Equation (4) shows how USL models the effect of the overheads due to contention(σ) and coherency(κ) in the overall execution time t, with sequential execution time of t_{seq}, on N cores, when $\frac{t_{seq}}{N}$ is the expected execution time on N cores in ideal case when the mentioned overheads are not present.

$$\begin{aligned} speedup \;&=\; \frac{t_{seq}}{t} \;=\; \frac{N}{1+\sigma(N-1)+\kappa N(N-1)} \Rightarrow \\ t \;&=\; \frac{t_{seq}}{N} \;+\; \sigma(N-1)\frac{t_{seq}}{N} \;+\; \kappa N(N-1)\frac{t_{seq}}{N} \end{aligned} \tag{4}$$

Based on Eq. (4), the term $\sigma(N-1)t_{seq}$ represents the overhead observed due to contention, and $\kappa N(N-1)t_{seq}$ is the overhead caused by coherency, assuming t_{seq} is the ideal execution time in this problem.

We need to keep in mind that there are cases where there is not enough work for all the cores to execute, causing the mentioned overheads to be a factor of the cores that are actually performing the work and not just all the available cores. For this reason, we adjust Eq. (4) by changing the total number of cores (N) to the number of cores that are actually being utilized (M).

Assuming that we are running our application on N cores, with a grain size equal to g, n_t tasks are being created, and M cores are being utilized. If $n_t < N$, M would be equal to n_t, otherwise $M = N$.

Equation (3) is then converted into:

$$execution\_time = \alpha \left\lceil \frac{n_t}{N} \right\rceil + \sigma(M-1)t_{max} + \kappa M(M-1)t_{max} + t_{max}. \tag{5}$$

For a balanced parallel for-loop, when $N = 1$ all the work will be assigned to the only available core, resulting in $w_{max} = p_s$. When $N > 1$, in the general case at most $\lceil \frac{n_t}{N} \rceil$ tasks would be assigned to a core. Therefore, a grain size of g would result in a maximum amount of work of $g\lceil \frac{n_t}{N} \rceil$ being assigned to one core, causing $w_{max} = g\left\lceil \frac{n_t}{N} \right\rceil$.

Also t_{max}, the time to execute w_{max} amount of work, can be estimated as $t_{max} = t_{seq}\frac{w_{max}}{p_s}$, where t_{seq} is the time it takes to run the total amount of work, p_s, sequentially. Equation (5) is then simplified into Eq. (6).

$$execution\_time = \alpha \left\lceil \frac{n_t}{N} \right\rceil + t_{seq}\frac{w_{max}}{p_s}(1 + \sigma(M-1) + \kappa M(M-1)) \tag{6}$$

We refer to (6) as our analytical model in the next sections.

3.1 Model Evaluation

In order to evaluate the model, we developed a simple parallel for-loop microbenchmark[1]. We refer to it as the for-loop microbenchmark. Each iteration consists of a while loop that makes sure the iteration lasts a certain amount of time (t_i). By setting $t_i = 1\mu sec$, and changing the number of iterations n, and chunk size c_s, we can see how the execution time changes when the microbenchmark is executed on different number of cores.

Having defined p_s as the total amount of work that has to be performed, for this microbenchmark, $p_s = t_i \times n$. Since $t_i = 1\mu sec$, then $p_s = n$. On the other hand, for this specific problem $w_{max} = t_{max}$, and $t_{seq} = p_s$.

The microbenchmark was then executed with different number of cores(N), number of iterations(n), and chunk sizes (c_s). For each n, c_s is changed from 1 to n in logarithmic scale. Each of these runs was executed on $1, 2, 3, ..., 8$ cores.

Using the collected data points for each problem size (p_s), the *optimize.curve_fit* package from *scipy* library in Python was used to fit our model to the collected data. Figure 1 shows the prediction results from the fitted model and the original data for $p_s = 100000$, on 8 cores.

The relative error of the prediction is calculated for each problem size based on Eq. (7), where p_k is the predicted value of the sample k, t_k is the true measured value, and K is the total number of samples.

$$Relative\_error = \frac{1}{K}\sum_{k=1}^{K} |1 - \frac{p_k}{t_k}| \tag{7}$$

[1] https://gist.github.com/shahrzad/b81e1eb252880aca48528d2de0bd1d10.

As discussed earlier, for a specific runtime system, the model parameters α and σ mostly depend on the system architecture and are expected to be constant for different problem sizes as long as they are executed on the same machine. Therefore, we suggest relying on the data collected from one problem size to find parameters α and σ. For this purpose, for problem sizes of 10000, 100000, 1000000, 10000000, 100000000, we used the parameters identified based on the data collected from one specific problem size, to estimate the execution time in terms of grain size and measure the relative error for the same problem size(Fig. 2a) and all other problem sizes(Fig. 2b).

Using the data collected for $p_s = 10000$ to estimate the model parameters generates higher prediction error on other problem sizes, but for other problem sizes we don't see a considerable change in prediction error. Since larger problem sizes require more data to be collected to cover the whole spectrum of grain sizes, $p_s = 100000$ was selected as a reasonable problem size to estimate the model parameters. Fitting our model to all the 512 data points collected for $p_s = 100000$, resulted in model parameters $\alpha = 2.42$ and $\sigma = 0.025$.

We suggest to run the for-loop microbenchmark on the desired system to run our parallel application on, for $p_s = 100000$, to estimate α and σ. Plugging the estimated parameters into Equation (6) would create the analytical model to be used for other balanced parallel for-loop applications executed on the same system.

Our experiments were run on a node consisting of two Intel(R) Xeon E5-2450 CPUs clocked at 2.1 GHZ amounting to a total of 16 cores and 48 GB RAM. Hyper-threading was turned off for the experiment. The versions of HPX used was 1.5.0.

3.2 Identifying the Optimal Range of Grain Size

The graph of execution time in terms of grain size in logarithmic scale, denoted as the *bathtub curve*, can be divided into three regions. We refer to these regions as the left side, the right side, and the flat regions of the graph. Figure 3 shows an example of the flat region of the execution time graph versus grain size in both linear and logarithmic scales. As it can be observed, the flat region contains a very small range of grain sizes.

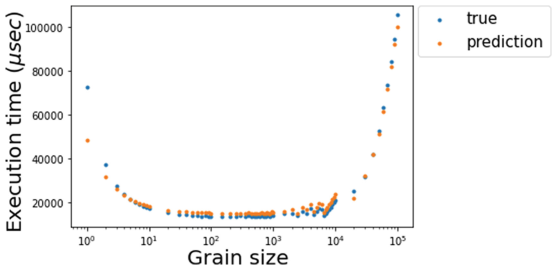

Fig. 1. The results of predicting the execution time based on the proposed model through curve fitting vs the real data for $p_s = 100000$, for 8 cores.

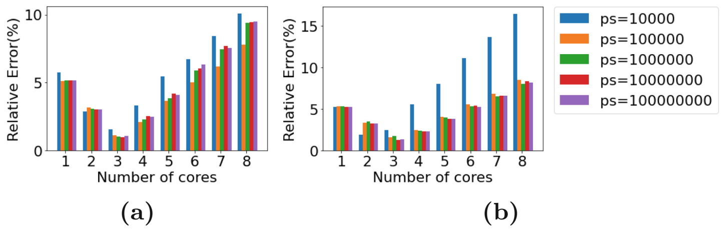

Fig. 2. The relative error of fitting the measured data for different problem sizes on (a) the same problem size and (b) other problem sizes, calculated for each number of cores.

3.2.1 Left Side of the Graph

In Equation (6), for small grain sizes the first term$(\alpha\lceil\frac{n_t}{N}\rceil)$ is the dominant factor while the second term$(t_{seq}\frac{w_{max}}{p_s}(1 + \sigma(M-1)))$ roughly stays constant. Likewise, for large grain sizes the second factor is the dominant factor.

In order to find the lower-bound of the range for which the execution time stays constant, we can assume that the second factor is constant in that region. Also, we can change N to M, knowing that our concern is on the left side of the graph, where n_t is definitely greater than the number of cores. Taking the derivative of the function based on the grain size leads to:

$$\frac{\partial execution\_time}{\partial g} = \frac{\alpha}{N}\frac{\partial n_t}{\partial g} = \frac{\alpha}{N}\frac{\partial(\frac{p_s}{g})}{\partial g} = \frac{\alpha}{N}p_s\frac{-1}{g^2}. \tag{8}$$

From (8), it can be observed that for the left side of the graph, the rate of change is negative, and it decreases as the grain size increases. Here we are looking for the value of the grain size for which the rate of change becomes very small (we introduce a threshold λ_b, where $0 < \lambda_b \ll 1$, for this purpose).

$$\frac{\alpha}{N}p_s\frac{1}{g^2} \leq \lambda_b \Rightarrow g \geq \sqrt{\frac{\frac{\alpha}{N}p_s}{\lambda_b}} \tag{9}$$

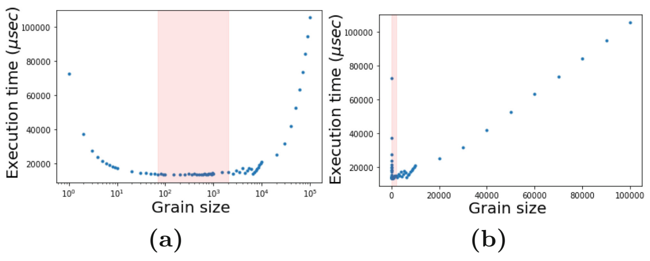

Fig. 3. The results of running the for-loop microbenchmark with $p_s = 100000$, on 8 cores in (a) logarithmic scale, and (b) linear scale.

Equation (9) can also be represented as shown in (10). This representation shows that when the ratio of the time it takes to execute one task to the total overhead of scheduling n_t tasks on N core, is greater than a threshold, we will end up in the flat region of the graph, close to the left side.

$$\frac{\alpha}{N}\frac{p_s}{g}\frac{1}{g} \leq \lambda_b \Rightarrow \frac{g}{\frac{\alpha}{N}n_t} \quad \geq \frac{1}{\lambda_b} \tag{10}$$

3.2.2 Right Side of the Graph

At the right side of the graph, the overhead of creating the tasks is negligible, since only a few tasks are being created and the associated overhead is not significant compared to the execution time. On this side, w_{max} and consequently t_{max} are the dominant factors. In general we can estimate w_{max} with $g\left\lceil\frac{n_t}{N}\right\rceil$. There are grain sizes for which $\left\lceil\frac{n_t}{N}\right\rceil$ is the same, but w_{max} would be different. For all the values of g that create the same $\left\lceil\frac{n_t}{N}\right\rceil$, as g increases, the difference between w_{max} and $\frac{p_s}{N}$ increases. This means that for a range of grain sizes with the same value of $\left\lceil\frac{n_t}{N}\right\rceil$, as we get closer to the end of the range, we are observing that a much bigger amount of work is assigned to the core with the maximum amount of work, which would result in a higher execution time.

In the general case, if we denote $\left\lceil\frac{n_t}{N}\right\rceil$ as k, then:

$$k-1 < \frac{n_t}{N} = \frac{\left\lceil\frac{p_s}{g}\right\rceil}{N} \leq k \Rightarrow \quad (k-1)N < \frac{p_s}{g} \leq kN. \tag{11}$$

If $k = 1$, then, $0 < \frac{p_s}{g} \leq N$ and therefore $\frac{p_s}{N} \leq g \leq p_s$. Otherwise, when $k > 1$ the following equation can be deduced.

$$\frac{p_s}{kN} \leq g < \frac{p_s}{(k-1)N} \tag{12}$$

Since $\left\lceil\frac{n_t}{N}\right\rceil = k$, and $w_{max} = g\left\lceil\frac{n_t}{N}\right\rceil = gk$, we can conclude for $k > 1$:

$$0 \leq w_{max} - \frac{p_s}{N} < \frac{1}{k-1}\frac{p_s}{N}. \tag{13}$$

And for $k = 1$, $w_{max} = g$, $n_t \leq N$, therefore $\frac{p_s}{N} \leq g \leq p_s$.

$$0 \leq w_{max} - \frac{p_s}{N} = g - \frac{p_s}{N} \leq (N-1)\frac{p_s}{N} \tag{14}$$

We define a new parameter *imbalance_ratio* as $(w_{max} - \frac{p_s}{N})/\frac{p_s}{N}$. Consequently,

$$\begin{cases} 0 \leq imbalance\_ratio \leq N-1 & \text{for} \quad k=1 \\ 0 \leq imbalance\_ratio < \dfrac{1}{k-1} & \text{for} \quad k>1 \end{cases} \tag{15}$$

Equation (15) shows that as the number of created tasks increases, while the number of tasks per core is the same, the imbalance factor decreases.

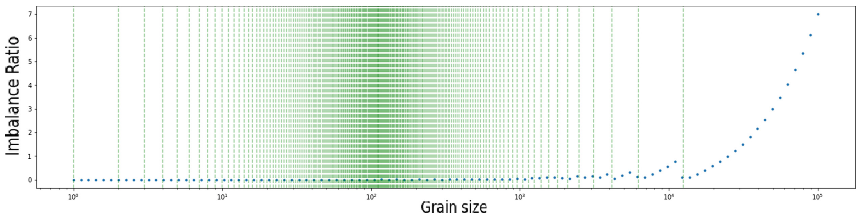

Fig. 4. The imbalance ratio calculated for different grain sizes for $p_s = 10000$, on 8 cores. At the area between each two green lines $k = \lceil \frac{n_t}{N} \rceil$ is constant.

Figure 4 shows how imbalance ratio changes for different grain sizes for $p_s = 10000$, on 8 cores. Each of the regions between two dashed green lines correspond to a specific value for $k = \lceil \frac{n_t}{N} \rceil$.

At each of the regions with $k > 1$, $\lceil \frac{n_t}{N} \rceil = k$, *imbalance_ratio* starts from 0 and approaches $\frac{1}{k-1}$ at the end of the region. When $k = 1$, *imbalance_ratio* increases linearly starting from 0 and reaching the maximum of $N - 1$ when $g = p_s$. As we move to larger grain sizes, $\lceil \frac{n_t}{N} \rceil$ decreases. We define a threshold, λ_s ($0 < \lambda_s < 1$), so that for *imbalance_ratio*s smaller than this threshold, the imbalance effect is not considered significant. Our goal would then become finding the maximum grain size that would generate a reasonable imbalance (*imbalance_ratio* $\leq \lambda_s$), to make sure we should stay in the flat region of the bathtub curve of execution time against grain size.

Equation (14) states that for grain sizes greater than $\frac{p_s}{N}$, *imbalance_ratio* increases linearly with grain size from 0 to $N - 1$. While for grain sizes smaller than p_s, the maximum *imbalance_ratio* depends on $k = \lceil \frac{n_t}{N} \rceil$. To ensure *imbalance_ratio* is smaller than or equal to a threshold (λ_s), first we search the grain sizes smaller than $\frac{p_s}{N}$. Since $0 < \lambda_s < 1$, and $k \geq 2$ in this region, there exists a k such that $\frac{1}{k-1} \leq \lambda_s$.

If there exists a k_{min}, for which *imbalance_ratio* $< \frac{1}{k_{min}-1}$, where $\frac{1}{k_{min}-1} \leq \lambda_s$, then $\forall k < k_{min}$, maximum value of *imbalance_ratio* would be greater than λ_s. To find the grain size that creates maximum *imbalance_ratio* of λ_s:

$$imbalance\_ratio \leq \lambda_s \;\Rightarrow\; \frac{1}{k-1} \leq \lambda_s \;\Rightarrow\; k_{min} = \left\lceil 1 + \frac{1}{\lambda_s} \right\rceil + 1 \tag{16}$$

Based on Equation (13), $g < \frac{p_s}{(k_{min}-1)N}$, and therefore:

$$g_{max} = \frac{p_s}{(k_{min}-1)N} - 1 = \frac{p_s}{(1 + \lceil \frac{1}{\lambda_s} \rceil)N} \tag{17}$$

If $g < g_{max}$, we can ensure that *imbalance_ratio* never exceeds λ_s. Since we already found a match at grain sizes smaller than $\frac{p_s}{N}$, checking the rest of grain sizes would not be necessary.

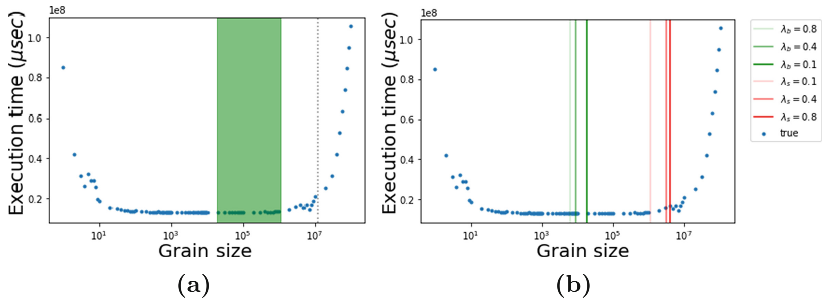

Fig. 5. (a)The identified range of grain sizes for $p_s = 100000000$ on 8 cores, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$. The gray dashed line represents the grain size where work is equally divided among the cores, $\frac{p_s}{N}$. (b) The effect of λ_b and λ_s on the borders of the identified region for minimum execution time.

3.3 Identifying the Range of Grain Sizes for Minimum Execution Time

In the previous section, we introduced a method to identify the lower-bound and upper-bound of grain sizes for which we expect to observe the minimum execution time. Integrating Eq. (9) and Eq. (17) suggests the following range for minimum execution time, Where $0 \leq \lambda_s \leq 1$, and $\lambda_b, \lambda_s \ll 1$:

$$\sqrt{\frac{(\frac{\alpha}{N})p_s}{\lambda_b}} \leq g \leq \frac{p_s}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil)N}. \tag{18}$$

Here λ_b indicates the slope of the graph at the left side of the graph where overhead of tasks is the dominant factor. Grain sizes smaller than $\sqrt{((\frac{\alpha}{N})p_s)/\lambda_b}$ would create a slope of more than λ_b. As for λ_s, a grain size greater than $\frac{p_s}{(1+\lceil \frac{1}{\lambda_s} \rceil)N}$ could generate an *imbalance_ratio* of greater than λ_s.

3.4 Locating the Flat Region of the Execution Time vs Grain Size Graph for the For-Loop Microbenchmark

In this section we used Eq. (18) to identify the flat region of the execution time vs grain size graph for the parallel for-loop microbenchmark. For this purpose, we set both λ_b and λ_s to 0.1.

In Fig. 5a the identified region for minimum execution time is shown in green, for $p_s = 10000000$, executed on 8 cores.

Selecting a greater value for λ_b would move the left border of the region to left, for a larger acceptable slope of change of execution time in terms of grain size. On the other hand, selecting a smaller value for λ_s would result in shifting the right border of the region to left, imposing a higher restriction on *imbalance_ratio*, as shown in Fig. 5b. λ_b and λ_s could be selected depending on how strict one wants to be in terms of slope of changes and imbalance ratio. In the meanwhile, based on our experiments we suggest $\lambda_b = 0.01$ and $\lambda_s = 0.1$ as reasonable values for

4 Related Work

Akhmetova et al. [1] utilized a system emulator to study the effect of task granularity in system performance. They also provide an algorithm to automatically aggregate tasks into larger tasks based on the calculated task granularity in order to improve the performance.

Grubel et al. also study the effect of the task size on performance of HPX applications[6]. They suggest using a number of performance metrics in order to identify the optimum grain size to improve the adaptivity at runtime.

In [13], the authors use thresholds to decide on whether to inline a task or not at runtime. The imposed threshold for task inlining on a specific architecture then converts into the problem to what portion of the execution time of the task should be spent for scheduling the task, so that it would be worth to be executed as a separate task. This is in compliance with our findings in this paper for λ_b, as shown in Eq. (10), where we suggest in order to land in the flat region of the execution time versus grain size graph, the ratio of the grain size over the scheduling overhead of one task on one core should be greater than the given threshold.

5 Conclusions and Future Work

In this paper we discussed the importance of task granularity on the achievable performance in AMTs. We offered an analytical model for execution time of a parallel application with a balanced for-loop, in terms of grain size and the number of cores. A for-loop microbenchmark was developed to validate this model and, a method has been provided to estimate the range of grain sizes to achieve the minimum execution time. At the next step, we suggested that we can use the developed for-loop microbenchmark with a fixed problem size to find the model parameters of a runtime system on a specific architecture. The identified parameters can then build the analytical model for arbitrary balanced parallel for-loop applications on the same machine.

For simplification and due to the nature of the for-loop microbenchmark we based our work on, we had ignored the κ parameter in USL model. For future work, we would like to study the effect of this parameter on both execution time and the upper-bound for the identified range.

Acknowledgment. The authors are grateful for the support of this work by the LSU Center for Computation & Technology and by the DTIC project: Phylanx Engine Enhancement and Visualizations Development (Contract Number: FA8075-14-D-0002/0007).

References

1. Akhmetova, D., Kestor, G., Gioiosa, R., Markidis, S., Laure, E.: On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In: 2015 IEEE International Conference on Cluster Computing, pp. 428–437. IEEE (2015)
2. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, 18–20 April 1967, pp. 483–485. ACM (1967)
3. Bennett, J., et al.: Asynchronous many-task runtime system analysis and assessment for next generation platforms. US Department of Energy, Sandia National Laboratories Report, Rep. no. SAND2015-8312 (2015)
4. Gao, G.R., Sterling, T., Stevens, R., Hereld, M., Zhu, W.: Parallex: a study of a new parallel computation model. In: 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1–6. IEEE (2007)
5. Grama, A., Kumar, V., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Pearson Education, Boston (2003)
6. Grubel, P., Kaiser, H., Cook, J., Serio, A.: The performance implication of task size for applications on the HPX runtime system. In: 2015 IEEE International Conference on Cluster Computing, pp. 682–689. IEEE (2015)
7. Grubel, P., Kaiser, H., Huck, K., Cook, J.: Using intrinsic performance counters to assess efficiency in task-based parallel applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1692–1701. IEEE (2016)
8. Gunther, N.J.: The practical performance analyst. iuniverse. com inc. Lincoln, Nebraska (2000)
9. Gunther, N.J.: What is Guerrilla Capacity Planning? Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-31010-5_1
10. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: 2009 International Conference on Parallel Processing Workshops, pp. 394–401. IEEE (2009)
11. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: a task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, p. 6. ACM (2014)
12. Kulkarni, A., Lumsdaine, A.: A comparative study of asynchronous many-tasking runtimes: Cilk, charm++, parallex and am++. arXiv preprint arXiv:1904.00518 (2019)
13. Wagle, B., Monil, M.A.H., Huck, K., Malony, A.D., Serio, A., Kaiser, H.: Runtime adaptive task inlining on asynchronous multitasking runtime systems. In: Proceedings of the 48th International Conference on Parallel Processing, pp. 1–10 (2019)