Self-organizing Core Allocation

Tobias Ziermann, Stefan Wildermann, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany

Abstract: This paper deals with the problem of dynamic allocation of cores to parallel applications on homogeneous many-core systems such as, for example, Multi-Processor System-on-Chips (MPSoCs). For a given number of thread-parallel applications, the goal is to find a core assignment that maximizes the average speedup. However, the difficulty is that some applications may have a higher speedup variation than others when assigned additional cores. This paper first presents a centralized algorithm to calculate an optimal assignment for the above objective. However, as the number of cores and the dynamics of applications will significantly increase in the future, decentralized concepts are necessary to scale with this development. Therefore, a decentralized (self-organizing) algorithm is developed in order to minimize the amount of global information that has to be exchanged between applications. The experimental results show that this approach can reach the optimal result of the centralized version in average by 98.95%.

1 Introduction

With the growing number of cores on MPSoCs [Bor07] the sequential execution of applications results in inefficient usage of processing resources. Therefore, future systems applications need to be able to efficiently exploit time-variant degrees of parallelism. The efficiency of running an application on multiple cores, i.e., the speedup compared to pure sequential execution, is depending on the degree of parallelism of an application. So, the question arises: how to assign the available cores to applications at any point of time? In this paper, we provide methods to determine assignments that increase the average speedup of all applications thereby increasing the total system throughput and consequently reducing average execution times.

Carrying out such assignments can generally be performed in two ways: centralized and decentralized. A centralized approach has the advantage of global knowledge which allows us, as we will see, to find an optimal assignment. The disadvantage is, however, that with shrinking transistor sizes the single point of failure cannot be tolerated. In addition, the steadily increasing number of cores on a chip will soon lead to unacceptable computation, monitoring and communication overheads.

Therefore, a decentralized approach is chosen motivated by the research area of Invasive Computing. Invasive computing [Tei08, THH+11] denotes a new programming paradigm for massively parallel MPSoCs where each application itself may dynamically and proactively quest for, occupy, and later release processors depending on the available degree of parallelism and state of the underlying processing resources, e.g., temperature, availability, load, permissions, etc. The goal of invasive computing is to provide room for dynamic system optimization by exploiting the knowledge of parallel algorithm and appli-

cation designers for resource allocation as well as to realize what is called *resource-aware programming*. In this paper, we consider just one particular problem of decentralized application control, namely the problem of understanding objectives and algorithms for the optimization of task, respectively thread allocation to processor cores.

This paper is organized as follows. In Section 2, the problem of dynamic task allocation is defined and put into context. Section 3 proposes an algorithm to find an optimal core assignment in quadratic time. Section 4 then introduces a new learning-based approach to assign cores decentrally at run-time to applications. Section 5 provides experimental results comparing the central and decentral algorithms in terms of convergence time and average speedup. Finally, conclusions and future work are given in Section 6.

2 Problem Description

In an informal simplified way, the problem we tackle can be described by the question: Which application allocates how many cores? Each application may have a different degree of parallelism. The locality of the cores is first neglected. The goal is to find core assignments with maximal average speedup. In the following, we motivate, then formalize and refine this informal problem description.

2.1 Related Work

The problem of scheduling parallel tasks has received a wide interest in research in the domain of high performance computing (HPC) [BDO08]. A reasonable amount of applications such as for example [BGT99] justify this interest. The application model including the speedup is the same as used in this paper, which backs up that our assumptions are realistic. One difference to the problem described here is that in previous approaches, the number of cores allocated to each application does not change at run-time [TWY92]. The possible performance gain when using task preemption justifies this assumption [BMW+04]. However, the novelty compared to all previous work is in the goal function. All existing work tackles as objective to minimize the makespan, or latest task completion time. As already described in the previous section, minimizing the average speedup may also minimize the average execution time. Furthermore, with the current and foreseen growing dynamics [Don12] and amounts of computation in HPC [AFG+10] the makespan is not computable as applications appear dynamically. Therefore, a new objective function is needed. Another reason for using the average speedup as objective function is that one of the key challenges in reaching exascale performance is the energy and power challenge [BBC+08]. This means, applications that achieve lower speedups with additional cores run less efficient and consume more energy per performance. When fulfilling the defined goal function all applications run most efficient. A further unique feature is the distributed nature of the presented self-organizing algorithm that will in our opinion get more and more important in future systems.

It has to be noted that there already exists a wide range of practical approaches [CLT13, GXWS11, CJ08]. Nevertheless, we see a gap between formal analysis and practical heuristics. We contribute to close this gap by providing algorithms first for simple assumptions.

2.2 Application and Speedup Model

For the analysis of optimal core allocation strategies, we assume that each application running on an MPSoC is a malleable program [Dow97]. This means that each application can run in parallel on any number of cores. Moreover, for each application under consideration, its speedup compared to running the application on one core is known in advance and doesn't change during run-time. Furthermore, we assume that all applications are known in advance, and become available for execution at the start and run forever. These assumptions on applications and speedup are only used to simplify the argumentation. Later, we will explain the consequences when loosening these assumptions.

Definition 1. Let the speedup $S_i(n)$ denote the quotient of the execution time when running application *i* on *n* cores relative to its execution time on one core:

$$S_i(n) = \frac{T_i(1)}{T_i(n)}, \tag{1}$$

where $T_i(n)$ is the execution time of application i when executed on n cores.

The speedup function $S_i(n)$ can be gathered by code analysis or by profiling. Moreover, cores may be only assigned to one application as whole. So, the speedup is defined for integers n from one to infinity. For later descriptions and proofs, the following definition is necessary.

Definition 2. Let delta speedup $\Delta S_i(n)$ denote the difference in speedup when application *i* runs on *n* cores instead of $n-1$ cores:

$$\Delta S_i(n) = \begin{cases} S_i(n) - S_i(n-1), & n > 0 \\ 0, & n = 0 \end{cases} \tag{2}$$

The speedup only considers applications on MPSoCs without hyperthreading or superscalar behavior. This has the following consequences: If the application is run on zero/one core the speedup is always zero/one ($S_i(0) = 0/S_i(1) = 1$). An application that is not parallelizable at all only achieves the minimal speedup of $S_i(n) = 1$ for $n \geq 1$. The theoretical maximum speedup is for applications that are fully parallelizable infinitely, thus obtaining a speedup of $S_i(n) = n$. In our speedup model, we assume that the speedup does not decrease when adding cores. This means the speedup is monotonic increasing with

$$0 \leq \Delta S_i(n) \leq 1. \tag{3}$$

The last assumption made is that the delta speedup $\Delta S_i(n)$ decreases with increasing number of cores, what means that the following inequality is fulfilled

$$\Delta S_i(n) \geq \Delta S_i(n+1). \tag{4}$$

This is a realistic assumption, meaning the more cores an application gets allocated the less efficient the application can make use of it.

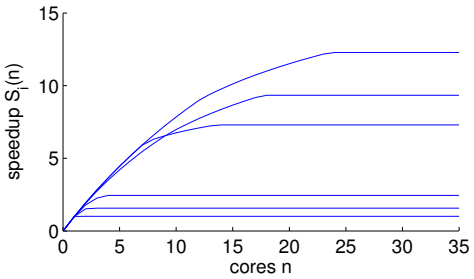


Figure 1: Example speedup characteristics generated using the Downey model.

Our speedup model is in line with state of the art models such as for example the widely used model [KBL+11, FR98] proposed by Downey [Dow97]. Figure 1 shows six example application speedup curves generated using the Downey model that are also used for later experiments.

2.3 Formal Definition of the Processor Allocation Problem

Problem 3 (Core Assignment Problem)**.** Given N applications numbered $i = \{1, 2, \ldots, N\}$ and a total number of C available cores. Let each application i be assigned a_i cores. As the number of available cores is limited to C,

$$\sum_{i=1}^{N} a_i \leq C. \tag{5}$$

The assignments a_i are gathered in a vector called assignment profile $\mathbf{a} = (a_1, a_2, \ldots, a_N)$. The goal is to find an assignment $\mathbf{a}$ that maximizes the sum of all speedups of all applications:

$$\max \left(\sum_{i=1}^{N} S_i(a_i) \right). \tag{6}$$

3 Central Assignment

In this section, we present an algorithm for optimally solving the assignment problem in case global knowledge is available.

Definition 4. Let $\mathbf{a}^{opt} = (a_1^{opt}, a_2^{opt}, ..., a_N^{opt})$ denote an *optimal assignment profile* if and only if the following proposition is fulfilled:

$$\sum_{i=1}^{N} S_i(a_i) \leq \sum_{i=1}^{N} S_i(a_i^{opt}),\ \forall \mathbf{a} = (a_1, a_2, \ldots, a_N).$$

Theorem 5. *There exists an optimal assignment profile* $\mathbf{a}^{opt}$ *with*

$$\sum_{i=1}^{N} a_i^{opt} = C \tag{7}$$

Proof. Due to the restriction in Equation (5), no more than C cores can be occupied, and the optimal assignment profile must obviously satisfy $\sum_{i=1}^{N} a_i^{opt} \leq C$. Of course, there might exist optimal assignments with $\sum_{i=1}^{N} \tilde{a}_i^{opt} < C$. However, as the speedup is monotonously increasing, we can conclude there will always exist an optimal assignment $\mathbf{a}^{opt}$ for which $\sum_{i=1}^{N} a_i^{opt'} = C$ holds. □

According to Theorem 5, it is sufficient to examine only those assignment profiles $\mathbf{a}$ with $\sum_{i=1}^{N} a_i = C$. This reduces the number of combinations that have to be tested for optimality from $(C+1)^N$ to

$$\sum_{i_{N-3}=1}^{C+1} (\cdots \sum_{i_2=1}^{i_3} (\sum_{i_1=1}^{i_2} (\sum_{i_0=1}^{i_1} i_0)) \dots).$$

Even with this reduced complexity, it is not feasible to test all combinations at run-time. Therefore, an algorithm to construct an optimal assignment profile is presented. With Definition 2, the problem of maximizing $\sum_{i=1}^{N} S_i(a_i)$ can be reformulated as follows:

$$\begin{aligned} S_i(j) &= \Delta S_i(j) + S_i(j-1) \\ &= \Delta S_i(j) + \Delta S_i(j-1) + S_i(j-2) \\ &= \Delta S_i(j) + \Delta S_i(j-1) + \cdots + \Delta S_i(1) + 0 \\ &= \sum_{x=1}^{j} \Delta S_i(x) \end{aligned}$$

We then can reformulate:

$$\sum_{a_i \in \mathbf{a}} S_i(a_i) = \sum_{a_i \in \mathbf{a}} \left(\sum_{x=1}^{a_i} \Delta S_i(x) \right) \tag{8}$$

Theorem 6. *Algorithm 1 denotes a central algorithm for determining an optimal assignment profile* $\mathbf{a}^{opt}$ *according to Definition 4.*

Proof. Given the set of delta speedups:

$$s_{all} = \{\Delta S_i(x) | x \in \mathbb{N} \wedge i = \{1, 2, \dots, N\}\}.$$

The sum of a subset $s_{assigned}$ of these delta speedups with C elements is greatest, when the smallest element of this subset $s_{assigned}$ is greater than the largest element of the remaining set $s_{remaining} = s_{all} \backslash s_{assigned}$. Such an assignment profile is constructed by taking always the greatest element of the set $s_{remaining}$ and inserting it into $s_{assigned}$. It is only then a valid assignment if only delta speedups of $\Delta S_i(x)$ are included in $s_{assigned}$, when all $\Delta S_i(y)$ with $y < x$ are already in $s_{assigned}$. This is indirectly met by the following argument. The greatest element of the set s_{all} is found by looking only at the N speedups with smallest x, because of Equation (4). This construction is carried out by Algorithm 1. □

Algorithm 1 Constructive algorithm for determining the optimal assignment a_i of core numbers to applications $i = 1, ..., N$.

1: **for** $i = 1$ **to** N **do**
2: $\quad a_i = 0$
3: **end for**
4: **for** $k = 1$ **to** C **do**
5: $\quad i = \arg \max_{j=1,...,N} (\Delta S_j(a_j + 1))$
6: $\quad a_i = a_i + 1$
7: **end for**

The worst-case execution time of the algorithm is $\mathcal{O}(N \cdot C)$, as C-times N elements have to be accessed and compared.

4 Decentralized Core Assignment

Based on the previous analysis, we present a decentralized core allocation strategy called *probabilistic delta speedup learning (PDSL)*. It follows the principle of the constructive algorithm described in Algorithm 1.

4.1 Algorithm

Each application itself increases or reduces the number of allocated cores in rounds, i.e., they claim cores. The idea is to increase the number of claimed cores with higher probability the higher the speedup gain is. Only increasing the number of assigned cores leads unavoidable to over-utilization. In this case, the applications with the least loss in speedup reduce the number of claimed cores with highest probability. The working principle is as follows: In each round, each application changes the number of claimed cores with a probability depending on $\Delta S_i(n)$, for which $0 \leq \Delta S_i(n) \leq 1$ holds. If an over-utilization occurs in the last round, each application reduces the number of claimed cores by one with probability $p_- = 1 - \Delta S_i(a_i)$. If no over-utilization occurred, the number of claimed cores is incremented by one (an additional core is requested) with probability $p_+ = \Delta S_i(a_i + 1)$.

A pseudocode description of PDSL is shown in Algorithm 2. The algorithms starts by initializing the number of claimed cores to zero. Note that this algorithm would also work

Algorithm 2 Probabilistic delta speedup learning (PDSL) of application i which dynamically increases or reduces the number of cores a_i claimed by this application.

```
1:  a_i = 0
2:  while (next step) do
3:    if (over-utilization) then
4:      if (ΔS_i(a_i) ≤ newRandom([0;1[)) then
5:        a_i = a_i − 1
6:      end if
7:    else
8:      if (ΔS_i(a_i + 1) > newRandom([0;1[)) then
9:        a_i = a_i + 1
10:     end if
11:   end if
12: end while
```

when starting with any other initial core assignment. The number of claimed cores is monotonously increased in each round as long as no over-utilization has occurred. This phase is called *attack time*. Then, the assignment may oscillate between over-utilization and under-utilization (*oscillation time*). Both during an increase and decrease of claimed cores it is ensured that with highest probability the largest delta speedups are chosen. Therefore, with highest probability, the optimal assignment according to Definition 4 is chosen. The greater the difference between the optimal and the second best assignment, the greater is the probability to reach and stay at the optimal assignment.

The constant oscillation ensures that the applications immediately react to changes in the system load. In this paper, we assume the overhead required for claiming and releasing cores is negligible. Depending on the underlying architecture, this overhead can be considerably high and a constant oscillation is not desirable. In this case, it is possible to stop the oscillation by simply not allowing applications to release cores. Then, after the attack time, the core assignment only changes when applications finish and, therefore, release cores. This behavior can be implemented by omitting lines 4 - 6 in Algorithm 2. This strategy has the drawback that it potentially leads to lower average speedups as disadvantageous claims cannot be reversed as quantified in Section 5.

4.2 Decentralized Detection of Over-utilization by Invasive Computing Mechanisms

Applications programmed by applying the *Invasive Computing* [Tei08, THH+11] principle, may use three operations. The *invade* operation explores and reserves cores available in the local neighborhood of the initial program. There exist techniques to extend custom MPSoCs by dedicated hardware modules, called *invasion controllers* [LNHT11], which perform this resource exploration. After having determined a proper set of cores, the program code is loaded onto the claimed resources through a so-called *infect* operation. Moreover, previously occupied resources can be freed by performing a release operation called *retreat*. The state of over-utilization can now be simply detected during the invade phase by the hardware controllers, and the invade operation terminates by indicating to the application that it was not able to claim successfully a number of requested processor

cores. The application can then communicate this over-utilization to the other applications in the system. This could be done by a multicast communication scheme, resulting in a communication overhead with a complexity of $\mathcal{O}(N)$. However, only a single bit needs to be communicated that is a small effort when implemented in dedicated hardware.

If using PDSL without release, no communication is necessary. In this case, the over-utilization information is not required and increasing of claimed cores will simply not be possible.

5 Experiments and Results

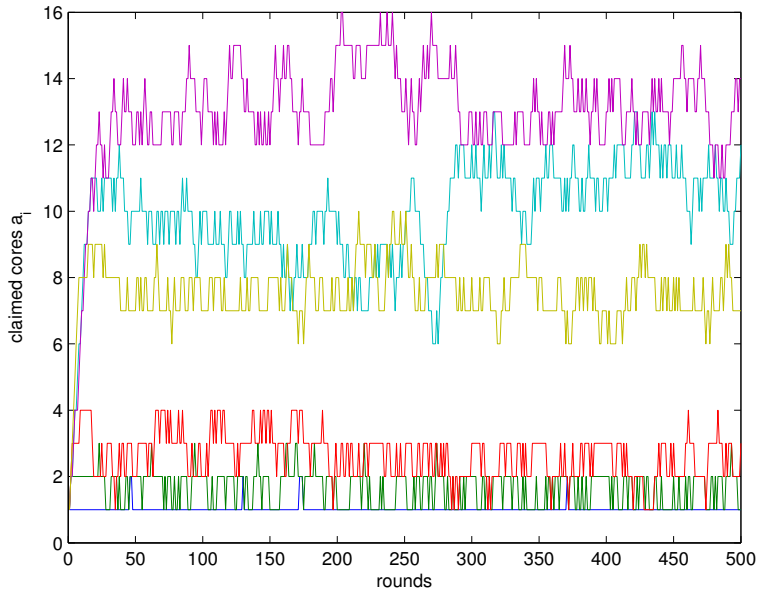
In this section, we compare quantitatively the mentioned assignment strategies. For the self-tuning decentralized approach in Algorithm 2 the following results are obtained by using a round-based simulation written in Java. The speedup used in the experiments is generated by the freely available job generator from [wwg]. The C program generates randomized speedups according to the Downey model. The resulting parameters imitate typical workloads similar to the SDSC Paragon logs and the CTC SP2 log. As already stated in Section 2.3, we do not consider arrivals other than at the beginning, thus, the arrival times are discarded. The case of over-utilization is handled as follows. Considering the scenario where applications dynamically claim cores in a distributed way, over-utilization can never occur, because the application trying to claim more cores than available will fail doing so. The experiments represent this behavior by using the results of the last valid assignment before the over-utilization.

Figure 2a shows the behavior of PDSL for six applications and 36 available cores. The simulation starts with one core per application. The applications that have the highest maximal speedup claim the highest number of cores. Until round 16, the number of claimed cores is increased (attack time), then the claimed cores oscillate around the optimum. This behavior is brought out very nicely by Figure 2b, where the speedup of the individual applications and the sum of the speedup is plotted over time.

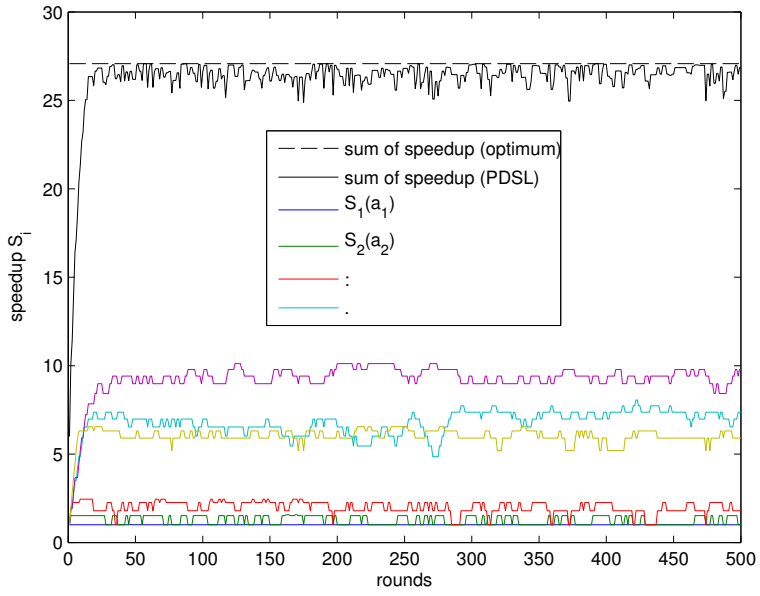
In the following, we will compare the speedup results of a) PDSL, b) the globally optimal value according to Section 3 and c) random assignments. The random assignments are generated by assigning one core to one application with equal probability until all available cores are assigned. The sum of speedups at the end of the attack time represent the result for using PDSL without releasing cores.

Figure 3 shows the sum of speedups for all three strategies for a system with $C = 36$ available cores and different numbers of applications. Until $N = 3$ applications, each application gets enough cores to exploit maximal parallelism by all three assignment strategies. For more applications, the assignment is not trivial and PDSL and the random assignment deviate from the optimal assignment. However, Figure 3 clearly indicates that the assignment generated by PDSL is very close to the optimal value. Even the results for using PDSL without release show a significant improvement over a random assignment.

We also conducted experiments with $C = 1024$. They show that the speedup value achieved by PDSL is very close to the optimal value with a relative error between PDSL and the optimum in average of 1.05%. In comparison, the average relative error between the random assignments and the optimum is 15.72% and this error increases with increas-



(a)



(b)

Figure 2: Number of claimed cores and speedup for number of applications $N = 6$ and available cores $C = 36$ over time using PDSL.

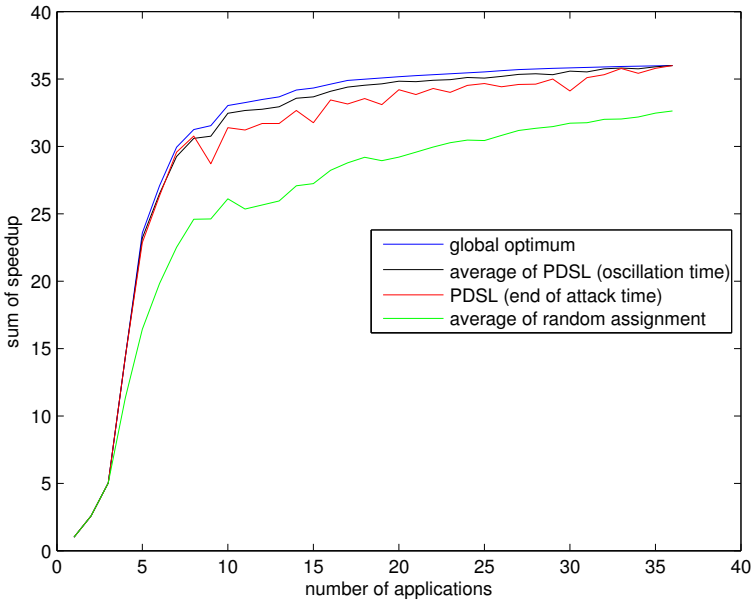


Figure 3: Sum of speedups for different number of applications running on an MPSoc with available cores $N = 36$.

ing number of cores. Using PDSL without release lies with 4,58% relative error in between the two.

To show the adaptability of PDSL also to changing applications, we ran the following experiment depicted in Figure 4: The first 500 rounds are the same as in simulation shown in Figure 2b. At round 500, ten additional applications are to be scheduled. From the experiments, we can clearly see that PDSL helps the applications to self-organize the core assignment in a close to optimal way even under changing applications. This property allows the algorithm to also work under changing speedup functions of the application. Therefore, the application can measure locally its actual speedup at run-time and improve accuracy of the speedup function.

6 Conclusions and Future Work

In this paper, we addressed the question of how cores can be assigned to malleable applications that have a defined speedup curve when running on more than one core. The restrictions on the speedup are in line with real systems, which is shown by an example. First, it is shown that for defined speedup functions, the assignment problem can be solved optimally using a centralized algorithm with global knowledge in $\mathcal{O}(N \cdot C)$. Second, the decentralized self-organizing algorithm PDSL is presented. There, the applications themselves learn how many cores to claim in order to maximize the overall average speedup. In contrast to the central approach, no direct communication is necessary between the appli-

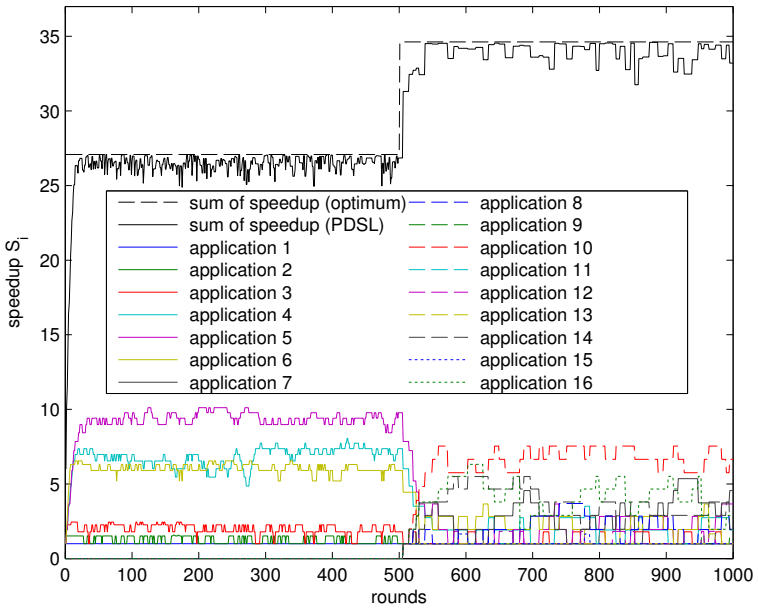


Figure 4: Speedup for $N = 6$ until round 500 then $N = 16$ and available cores $C = 36$ over time using PDSL.

cations, respectively applications. The only requisite is that the cores are aware of a global overload situation. Experimental results over up to $C = 1000$ cores and up to $N = 1000$ applications show that the average speedup obtained is in average 98.95% of the optimum.

The goal of future work will be to lower the abstraction level and consider the above approach on real machines that allow run-time assignment of cores. This will help answering the following open questions: (1) How high is the overhead for claiming and releasing cores? With this information it is possible to judge the penalty for oscillation. (2) What are the communication costs? Depending on the communication costs it might be necessary to find solutions not using the global information of over-utilization. Additionally, it is possible to make a comparison between a centralized and decentralized approach. (3) How accurate is the precalculated speedup and can we use run-time information to improve the accuracy?

References

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[BBC+08] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[BDO08] E.K. Burke, M. Dror, and J.B. Orlin. Scheduling malleable tasks with interdependent processing rates: Comments and observations. *Discrete Applied Mathematics*, 156(5):620–626, 2008.

[BGT99] P.-E. Bernard, T. Gautier, and D. Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of 13th International and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP*, pages 638–644, apr 1999.

[BMW+04] J. Błażewicz, M. Machowiak, J. Węglarz, M.Y. Kovalyov, and D. Trystram. Scheduling Malleable Tasks on Parallel Processors to Minimize the Makespan. *Annals of Operations Research*, 129:65–80, 2004.

[Bor07] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[CJ08] Jian Chen and L.K. John. Energy-aware application scheduling on a heterogeneous multi-core system. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 5 –13, sept. 2008.

[CLT13] Ya-Shu Chen, Han Chiang Liao, and Ting-Hao Tsai. Online Real-Time Task Scheduling in Heterogeneous Multicore System-on-a-Chip. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):118 –130, jan. 2013.

[Don12] J. Dongarra. talk: On the Future of High Performance Computing: How to Think for Peta and Exascale Computing. SCI Institute, University of Utah, 2012.

[Dow97] A.B. Downey. A parallel workload model and its implications for processor allocation. In *High Performance Distributed Computing. Proceedings. The Sixth IEEE International Symposium on*, pages 112–123, aug 1997.

[FR98] D. Feitelson and L. Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, pages 1–24, London, UK, 1998. Springer-Verlag.

[GXWS11] Xiaozhong Geng, Gaochao Xu, Dan Wang, and Ying Shi. A task scheduling algorithm based on multi-core processors. In *Mechatronic Science, Electric Engineering and Computer (MEC), 2011 International Conference on*, pages 942 –945, aug. 2011.

[KBL+11] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel. DistRM: distributed resource management for on-chip many-core systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 119–128, New York, NY, USA, 2011. ACM.

[LNHT11] V. Lari, A. Narovlyanskyy, F. Hannig, and J. Teich. Decentralized dynamic resource management support for massively parallel processor arrays. In *Application-Specific Systems, Architectures and Processors (ASAP), IEEE International Conference on*, pages 87–94, sept. 2011.

[Tei08] J. Teich. Invasive Algorithms and Architectures. *it - Information Technology*, 50(5):300–310, 2008.

[THH+11] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, Schröder-Preikschat, and G. Snelting. Invasive Computing: An Overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.

[TWY92] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 323–332, 1992.

[wwg] webpage workload generator. `http://www.cs.huji.ac.il/labs/parallel/workload/models.html#downey97`.