Game-Theoretic Analysis of Decentralized Core Allocation Schemes on Many-core Systems

Stefan Wildermann, Tobias Ziermann, Jürgen Teich
University of Erlangen-Nuremberg, Germany
{stefan.wildermann, tobias.ziermann, teich}@fau.de

***Abstract*—Many-core architectures used in embedded systems will contain hundreds of processors in the near future. Already now, it is necessary to study how to manage such systems when dynamically scheduling applications with different phases of parallelism and resource demands. A recent research area called *invasive computing* proposes a decentralized workload management scheme of such systems: applications may dynamically claim additional processors during execution and release these again, respectively. In this paper, we study how to apply the concepts of invasive computing for realizing decentralized core allocation schemes in homogeneous many-core systems with the goal of maximizing the average speedup of running applications at any point in time. A theoretical analysis based on game theory shows that it is possible to define a core allocation scheme that uses local information exchange between applications only, but is still able to provably converge to optimal results. The experimental evaluation demonstrates that this allocation scheme reduces the overhead in terms of exchanged messages by up to 61.4% and even the convergence time by up to 13.4% compared to an allocation scheme where all applications exchange information globally with each other.**

I. Introduction

Since 2003, the semiconductors' trend of increasing the clock frequency of processors has shifted towards increasing the number of processing cores on single chips due to physical issues concerning heat, power consumption, and leakage problems [1]. In a few years, many-core architectures will even contain hundreds of processing elements [2]. The consequence of this development is that it becomes more and more important to exploit the parallelism inherent in applications as, e.g., known from signal and image processing, to achieve a speedup in execution time. Only in this way, it is possible to cope with the increasing complexity and computational requirements of embedded systems with highly dynamic applications as, e.g., smartphones and smart cameras. A further challenge is that the workload is typically not fixed anymore and several applications may be dynamically activated and concurrently executed, requiring different amounts of processing cores. This raises the question how to assign the available cores to the parallelizable applications at any point in time? While the single applications are competing for cores, one main objective from the designer's point-of-view is to provide a fair allocation of cores to the competing applications such that their average speedup is maximized. Also, as a side effect, this increases the total system throughput as, e.g., signal/image processing applications are typically executed periodically. Only *run-time management* mechanisms are able to deal with such dynamic scenarios in embedded many-core systems. For being able to cope with hundreds or thousands of processors, it is also necessary to consider the scalability.

978-3-9815370-0-0/DATE13/©2013 EDAA

A recent research area called *invasive computing* [3] investigates the question of how to provide decentralized management schemes for future embedded many-core systems. The basic idea is that an *invasive application* may dynamically explore and automatically request a certain amount of processing cores during its execution, and release them again for processing software portions with a reduced degree of parallelism. This is achieved through two basic programming constructs called *invade* and *retreat*, respectively.

In this paper, standard game theory is applied to analyze how these concepts may be applied to provide a decentralized core allocation scheme for maximizing the average speedup of multiple applications. For these theoretical results, we focus first on homogeneous many-core systems. The analysis shows that it is necessary to synchronize *invade* and *retreat* requests of applications to enable the exchange of processing cores. The applications provably converge to an optimal core allocation when globally exchanging information. Our analysis furthermore reveals that an allocation scheme can be obtained that even then provably converges to optimal solutions when information is only exchanged locally between applications on adjacent processing cores. The results of the experimental evaluation demonstrate that, by applying this local communication scheme, the number of exchanged messages and the convergence time can be reduced considerably compared to an allocation scheme where information is exchanged globally between all applications. While the investigation is fairly motivated by invasive computing, the results are also applicable for any other run-time management approach, e.g, relying on multi agent systems (MASs).

II. Related Work

For embedded systems which demand high scalability and reliability, there are very good reasons (*cf.* [4]) to provide decentralized system management instead of using a Centralized Manager (CM) which cares for application mapping: No single point of failure, and avoiding to build up a hot-spot and bottleneck at the processor running the CM. Moreover, the communication and/or computation overhead required for monitoring and managing the applications and the system can be reduced. Consequently, several approaches have been proposed for management of multi-processor systems based on MASs. In [4], *cluster agents* are introduced which are responsible for managing a sub-set (cluster) of processors. Whenever an application is started, cluster agents can negotiate via *global agents* to map this application and, if necessary, to rearrange the clusters to meet the application's execution constraints. The authors of [5] provide a MAS approach tailored to the problem of core allocation as considered in our paper. Here, an agent will be spawned together with a newly started application.

Then, the agent will claim cores and negotiate with agents on neighbor cores to possibly gather further cores for then executing the application. The authors provide a negotiation protocol, which is then empirically evaluated via simulation. However, the theoretical and experimental results of our paper show that negotiation which is performed locally between applications may result in sub-optimal core allocations.

The authors of [6] take a more formalized approach to designing resource management mechanisms. They describe the problem of resource allocation in reconfigurable multi-core architectures as a *minority game* [7]. However, their purpose is to provide an abstract model of the problem to empirically evaluate their algorithm, instead of theoretically analyzing the model with the goal to develop optimal strategies.

In contrast to the above works, this papers investigates core allocation schemes for many-cores by using game theory. This enables to establish a formal proof of sub-optimality and optimality of the investigated allocation schemes. We subsequently derive proper run-time core allocation schemes based on these theoretical results.

III. Preliminaries and Problem Definition

For our theoretical analysis, a set of N applications is given, where each application is uniquely identified by an integer value $i \in \{1, ..., N\}$. When an application is started, it is assigned to a single free core of the many-core system. Any number of processing cores can be allocated to an application. If the application is parallelizable, it can spread its workload on allocated cores so that the time for executing the application is assumed to decrease. Let $T_i(n)$ denote the (expected) execution time of application i when running on n cores. Then, the *speedup* of execution is defined as

$$S_i(n) = T_i(1)/T_i(n). \tag{1}$$

The *speedup gradient* denotes the increase of speedup achievable when increasing the number of allocated cores by one. It is defined as

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

In standard speedup models, like the one proposed by Downey [8], it holds that $S_i(0) = 0$ and $S_i(1) = 1$ and the following properties hold:

- The gradient of the speedup function is limited to $0 \leq \Delta S_i(n) \leq 1$.
- The speedup function is monotonically increasing. However, its gradient is not increasing (no inflection point) $\Delta S_i(n) \leq \Delta S_i(n-1)$.

According to [9], above assumptions also hold when including parallelization overhead into the speedup model. For our analysis, we make some architectural assumptions: We consider homogeneous many-core system with $C \in \mathbb{N}$ processing cores. Moreover, applications use their allocated cores exclusively, and are able to exchange messages with each other via appropriate communication architectures like distributed shared memory or network-on-chips.

A. Problem Definition

With this notation, it is now possible to formalize the core allocation problem tackled in this paper.

Definition 1. *A vector* $\vec{C} = (C_1, ..., C_N)^{\mathrm{T}}$ *is called a* core allocation *and assigns each application* i *a number of cores* C_i. *For a* feasible core allocation, $\sum_{i=1}^{N} C_i \leq C$ *must hold.*

The goal is to provide a decentralized scheme for determining feasible core allocations that maximize the average speedup of all N applications being executed. This objective is expressed by

$$\text{maximize } \eta \cdot \sum_{i=1}^{N} S_i(C_i), \tag{3}$$

where the normalizing factor $\eta = \frac{1}{N}$ will be ignored in the following.

B. Optimal Core Allocation

The sum of speedups in Eq. (3) can be rewritten equivalently using the speedup gradients from Eq. (2) as follows:

$$\begin{aligned} S_i(C_i) &= \Delta S_i(C_i) + S_i(C_i - 1) \\ &= \Delta S_i(C_i) + \Delta S_i(C_i - 1) + S_i(C_i - 2) \\ &= \Delta S_i(C_i) + \Delta S_i(C_i - 1) + ... + \Delta S_i(1) + 0 \\ &= \sum_{n=1}^{C_i} \Delta S_i(n). \end{aligned} \tag{4}$$

Likewise, the overall sum of speedups can be rewritten by using the gradients:

$$\sum_{i=1}^{N} S_i(C_i) = \sum_{i=1}^{N} \sum_{n=1}^{C_i} \Delta S_i(n). \tag{5}$$

Now, it is possible to derive two important characteristics of an optimal core allocation $\vec{C}^{\text{opt}} = (C_1^{\text{opt}}, ..., C_N^{\text{opt}})^{\mathrm{T}}$:

1) W.l.o.g it allocates exactly C cores[1]:

$$\sum_{i=1}^{N} C_i^{\text{opt}} = C. \tag{6}$$

2) The sum of speedups of Eq. (5) can not be increased when exchanging cores between applications. This can be formalized according to

$$\min_{i=1,...,N} \{\Delta S_i(C_i^{\text{opt}})\} \geq \max_{j=1,...,N} \{\Delta S_j(C_j^{\text{opt}} + 1)\}. \tag{7}$$

meaning that the minimal speedup loss is equal or bigger than the maximal speedup gain when cores would be transferred between any applications.

C. Game-Theoretical Formulation of Core Allocation

In the following section, a game-theoretical formulation of the decentralized core allocation problem is presented that serves as a basis for subsequently being able to derive and compare different strategies. In a fully decentralized system as considered here, each application is assumed to request and also allocate a number of cores by itself. In this case, each application may be regarded as one player of a game with a set of actions to allocate and deallocate cores. The outcome of

[1]Note that an application with C_i allocated cores does not necessarily have to execute programs on each of these cores, particularly when $\Delta S_i(C_i) = 0$. However, for our theoretical analysis, we assume that also allocated idle cores are blocked for the other applications.

this game is a core allocation $\vec{C}$. A player's gain, called *utility*, depends on this outcome and is defined by a utility function $u_i(\vec{C})$. The goal of each player is to maximize its utility.

IV. Analysis of the Local Usage of invade/retreat

In a game-theoretical formulation, the goal of optimizing the average speedup can be expressed by the following utility function:

$$u_i(\vec{C}) = \begin{cases} \sum_{j \in \mathcal{N}_i} S_j(C_j), & \text{if } \sum_{k=1}^{N} C_k \leq C \\ 0, & \text{else} \end{cases}, \tag{8}$$

where $\mathcal{N}_i$ denotes those applications that application i is aware of. The utility shows that application i tries to maximize the average speedup of all applications in this *neighborhood*, as long as the amount of claimed cores stays within the number C of available cores. Fig. 1 illustrates some neighborhood relations: (a) in a purely local setup, each application is only aware of itself. The neighborhood can be extended by including neighborhood relations with more neighbors, *cf.* (b) and (c). In a global setup, the neighborhood includes all applications, *cf.* (d). Formally, the relations may be expressed by a neighborhood graph $G(V, E)$ where each application i is represented by a vertex $v_i \in V$ and each undirected edge $(v_i, v_j) \in E$ specifies which applications v_i and v_j may exchange information.

In the most basic form of the core allocation game, each player, resp. application has the following options according to the idea of invasive computing [3]:

- `invade(n)`: try to allocate n additional cores.
- `retreat(n)`: deallocate n claimed cores.

For the theoretical investigations, we analyze the *equilibria* of this game. An equilibrium is reached, possibly after repeatedly playing the game, when no player has any gain if changing its strategy. Thus, equilibria represent the possible outcomes of decentralized core allocation. In the following, we focus on *Nash equilibria* of the game.

Definition 2. *A core allocation game is in a* Nash Equilibrium *iff the players have reached a core allocation $\vec{C}^*$ for which it holds that*

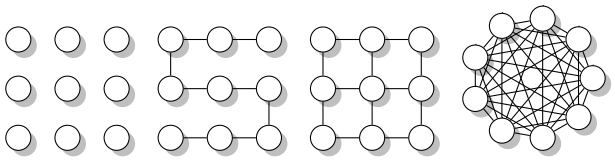
$$u_i(\vec{C}^*_{-i}, C_i) \leq u_i(\vec{C}^*_{-i}, C^*_i), \quad \forall C_i \in [0, C], \ \forall i, \tag{9}$$

where $\vec{C}^*_{-i} = (C^*_1, ..., C^*_{i-1}, C^*_{i+1}, ..., C^*_N)$ contains the core allocations of all applications except those of application i.

This definition states that a Nash equilibrium is an outcome $\vec{C}^*$ of the game where no player achieves a better utility when unilaterally choosing a different number of cores $C_i \neq C^*_i$ while all other players $j \neq i$ keep their cores C^*_j.

Theorem 1. *Not every equilibrium of the core allocation game corresponds to an optimal core allocation $\vec{C}^{opt}$.*

Proof: We will show that each core allocation $\vec{C}^*$ with $\sum_{j=1}^{N} C^*_j = C$ is a Nash equilibrium. While the optimal core allocation fulfills this condition, also does a multitude of sub-optimal allocations. Thus, proving this statement suffices as proof of the theorem as it shows that sub-optimal allocations may also be equilibria of the game.



(a) none (b) chain (c) mesh (d) full

Fig. 1. Examples of possible topologies of neighborhood relations where none, some, or all applications (vertices) exchange information with each other (acc. to the edges).

Take any core allocation $\vec{C}^*$ fulfilling this condition. We will now analyze the change of each player's utility, when it chooses the action `invade(n)` or `retreat(n)`, respectively.

If player i chooses to play `invade(n)`, it claims $C^*_i + n$ cores. This means that

$$\sum_{j=1}^{N} C^*_j + n > C \tag{10}$$

which is an infeasible core allocation and implies that utility $u_i(\vec{C}^*_{-i}, C^*_i + n) = 0 < u_i(\vec{C}^*)$ according to Eq. (8).

If player i chooses to play `retreat(n)`, it claims $C^*_i - n$ cores. As the utility corresponds to the sum of speedups, we get as a result

$$u_i(\vec{C}^*_{-i}, C^*_i - n) = \sum_{j \in \mathcal{N}_i} S_j(C^*_j) - \underbrace{(S_i(C^*_i) - S_i(C^*_i - n))}_{\geq 0}. \tag{11}$$

As the speedup is monotonously increasing, the decision of reducing the number of cores also leads to no increase of the utility so that $u_i(\vec{C}^*_{-i}, C^*_i - n) \leq u_i(\vec{C}^*_{-i}, C^*_i)$.

Thus, neither `invade()` nor `retreat()` improves the utility. ■

In the following, we show that by extending the set of possible actions it is possible to reach and guarantee optimality. The additional option is the following:

- `transfer(n, j)`: specifies a synchronized retreat/invade which basically means that the calling player retreats from n cores while simultaneously player j invades these n cores.

Optimality guarantees for this game, however, depend on the neighborhood relations.

Theorem 2. *When including option `transfer()`, each outcome is optimal for neighborhood $\mathcal{N}_i = \{1, 2, ..., N\}$, $\forall i$ (*cf. *Fig. 1 (d)). However, if $\exists i$ for which $\mathcal{N}_i \subset \{1, 2, ..., N\}$ (e.g., Fig. 1 (a)-(c)), results may also be sub-optimal.*

Proof: Consider any core allocation $\vec{C}$ which is not optimal. For such an allocation, we can conclude from Eq. (7) that there is a $\Delta S_i(C_i)$ and there is a $\Delta S_j(C_j + 1)$ for which it holds that

$$\Delta S_i(C_i) < \Delta S_j(C_j + 1). \tag{12}$$

- If the neighborhood of every player i is $\mathcal{N}_i = \{1, 2, ..., N\}$, player i playing `transfer(1, j)` results in a sum of speedups as:

$$\underbrace{\sum_{k \in \mathcal{N}_i}^{N} S_k(C_k)}_{u_i(\vec{C})} + \underbrace{\Delta S_j(C_j + 1) - \Delta S_i(C_i)}_{>0} > u_i(\vec{C}). \tag{13}$$

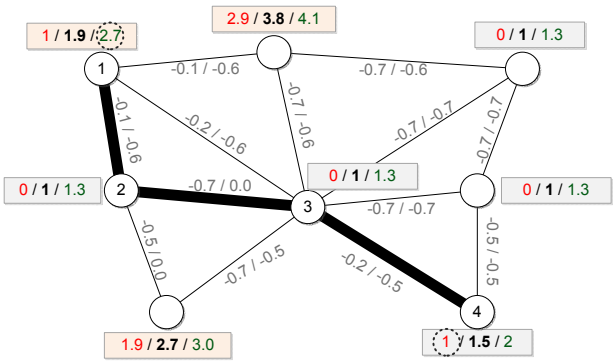


Fig. 2. Illustration of applications with neighborhood relations (indicated by the edges). The labels at each vertex v_i specify the speedup of the current core allocation, as well as the speedup when allocating or deallocating a core by giving $S_i(C_i-1)$ / $S_i(C_i)$ / $S_i(C_i+1)$. Each edge (v_i, v_j) is labeled with $\Delta_{i\to j}$ / $\Delta_{i\leftarrow j}$ where $\Delta_{i\to j}$ and $\Delta_{i\leftarrow j}$ denote the speedup increase when transferring one core from i to j and j to i, respectively.

This means that $\vec{C} \neq \vec{C}^{\text{opt}}$ is no Nash equilibrium.

- If, however, $\mathcal{N}_i \subseteq \{1, ..., j-1, j+2, ...N\}$, player i is not aware of j and will not play `transfer(1,j)`. As a consequence, also sub-optimal results may be obtained.

■

Example: Fig. 2 illustrates an example neighborhood graph illustrating the latter case. The labels at the applications specify the speedup of the current core allocation. Each edge (v_i, v_j) is labeled with the speedup increases obtainable when exchanging one core between i and j. Note that in this scenario, despite application 3 being aware of all other applications, all edge labels are equal or less than 0. Thus, no speedup increase is possible by locally transferring cores, and no application has an incentive to execute `transfer()`*. However, when transferring cores from application 4 to application 1, it would be possible to increase the accumulated speedup by $+0.3$ (what also leads to an increase of the average speedup).*

V. Communication Scheme for Provably Reaching Optimal Core Allocations

As a solution to the above problem, this section provides a communication scheme with which it is possible to provably reach optimal core allocations, even if there are applications i with local neighborhoods $\mathcal{N}_i \subset \{1, 2, ..., N\}$. The idea of our communication scheme is to use the local neighborhood relations for establishing an *overlay network* to globally communicate information. Each player i keeps track of that player ind_i that has the highest speedup gradient known to it. Furthermore, each player communicates this information to its neighborhood $\mathcal{N}_i$. Following this scheme, ind_i denotes either the player i itself or the player ind_j known to any of its neighbors $j \in \mathcal{N}_i$, depending on which one has the maximal value:

$$ind_i = \underset{\substack{l \in \{i\} \cup \\ \{ind_j | j \in \mathcal{N}_i\}}}{\arg\max} \{\Delta S_l(C_l+1)\}. \quad (14)$$

Consequently, in this scenario, the set of known players $\mathcal{N}_i'$ used for utility computation (thus, replacing $\mathcal{N}_i$ in Eq. (8)) is given as the union of i's neighbors and ind_i:

$$\mathcal{N}_i' = \mathcal{N}_i \cup \{ind_i\}. \quad (15)$$

By doing this, it is possible to communicate the maximal known speedup gradient in the network of players by forwarding it to all neighbors, which again forward it to their neighbors, and so on. The preliminary for this approach is, however, that each player i is connected with every other player j possibly transitively over several neighborhoods. This means that the neighborhood graph $G(V, E)$ has to be connected, i.e.,
$\forall i, j, \exists k_1, k_2, ..., k_n$:

$$(v_j, v_{k_1}), (v_{k_1}, v_{k_2}), ..., (v_{k_{n-1}}, v_{k_n}), (v_{k_n}, v_i) \in E. \quad (16)$$

Theorem 3. *Each Nash equilibrium is an optimal solution when applying the communication scheme.*

Proof: First, we prove that an optimal core allocation $\vec{C}^{\text{opt}}$ is a Nash equilibrium. For any player i, player ind_i is known to be the player with biggest speedup gain $\Delta S_{ind_i}(C_{ind_i}+1)$. However, for an optimal core allocation, we can conclude from Eq. (7) that

$$\Delta S_i(C_i^{\text{opt}}) \geq \Delta S_{ind_i}(C_{ind_i}+1) \quad (17)$$

and all players i have no incentive to trade cores.

Second, we prove that any non-optimal core allocation $\vec{C}$ is no Nash equilibrium. In a non-optimal allocation, it holds that there is at least one $\Delta S_j(C_j+1) > \Delta S_i(C_i)$, acc. to Eq. (7). From Eq. (16), there exists a path over $k_1, ..., k_n$ connecting i and j. Player j's speedup gradient is then forwarded via this path according to the communication scheme that is formalized in Eq. (14). This means, that, after the information is forwarded via $j, k_1, ..., k_n$, the known players of i eventually are $\mathcal{N}_i' = \mathcal{N}_i \cup \{j\}$. As a consequence, the following holds

$$u_i(\vec{C}) < \underbrace{\sum_{k \in \mathcal{N}_i'} \sum_{n=1}^{C_k} \Delta S_k(n)}_{u_i(\vec{C})} + \underbrace{\Delta S_j(C_j+1) - \Delta S_i(C_i)}_{>0} \quad (18)$$

and player i has an incentive to play `transfer(1,j)` to increase the utility accordingly by $\Delta S_j(C_j+1) - \Delta S_i(C_i)$.

■

Example: Consider the example from Fig. 2. Application 1 has a speedup gradient $\Delta S_1(C_1+1) = 0.8$. This information is communicated, e.g., via 2 and 3 to application 4, which has a speedup loss of $\Delta S_4(C_4) = 0.5$. So, application 4 will transfer cores to application 1 to increase the overall speedup by $+0.3$.

The algorithm implementing the above communication scheme is illustrated by the flowchart in Fig. 3. Each application i starts with an initial invasion where cores are claimed until it is unsuccessful. As a consequence, we reach a core allocation $\vec{C}$ where $\sum_{j=1}^{N} C_j = C$. As already mentioned before, the applications don't actually have to execute threads on the invaded cores, but still block them for other applications.

In the next step, requests from other applications are processed if any are available. Such requests come from neighboring applications and specify an application j and its (expected) speedup gain $\Delta S_j(C_j+1)$ for retrieving one additional core. Now, this request is compared to the own speedup gain $\Delta S_i(C_i+1)$. If the request is smaller, the application sends a request on its own. Otherwise, if the own

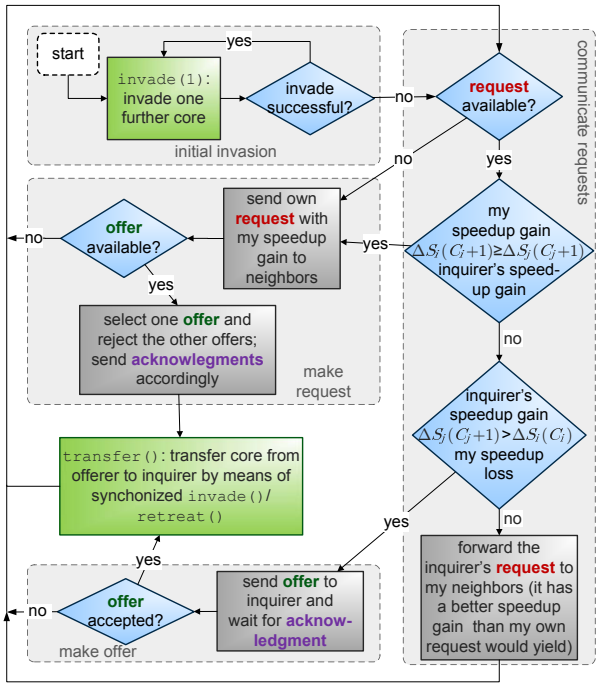


Fig. 3. Flowchart illustrating the communication scheme.

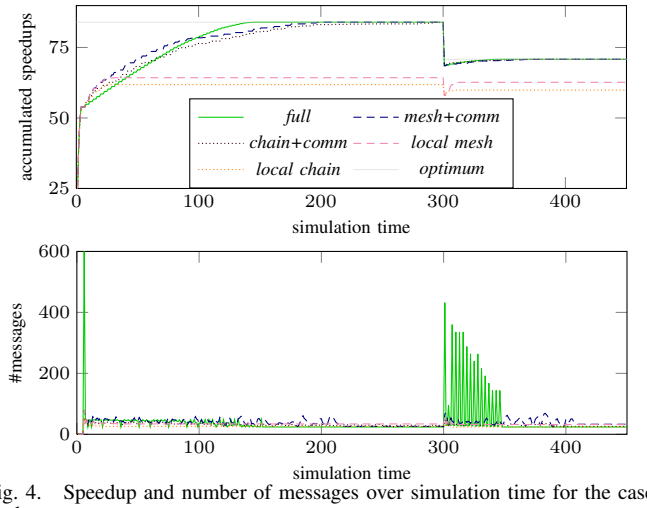


Fig. 4. Speedup and number of messages over simulation time for the case study.

speedup loss $\Delta S_i(C_i)$ is smaller than the speedup gain of j, application i sends an offer to j via the network of neighboring applications. When it receives an accepting acknowledgment, it executes `transfer()`. In a possible implementation of this behavior, cores at the common boundary of adjacent applications can be handed over subsequently according to the neighborhood relations. For example, consider Fig. 2. Even if applications 4 and 1 are not adjacent, application 4 can transfer cores to application 1 by exchanging cores between the highlighted path of applications. So, cores are transferred locally between applications which are located adjacent on the chip along the path that was previously used to send the speedup information from 4 to 1.

VI. Experimental Evaluation

In the following experiments, we compare the optimality and convergence overhead of the fully connected (called *full*), mesh, and chain neighborhood topologies, *cf.* Fig. 1. For the mesh and chain topologies, we test the scheme of local information exchange from Section IV (called *local mesh* and *local chain*), as well as the communication scheme introduced in Section V (called *mesh+comm* and *chain+comm*). All experiments are performed by simulating the execution of the algorithm from Fig. 3 synchronously for each application. Thus, we are able to measure the speedups and exchanged messages over simulation time ($\widehat{=}$ rounds playing the allocation game).

A. Case Study

A many-core system containing 100 cores serves as a case study. Initially, 25 applications are concurrently started. Fig. 4 illustrates their accumulated speedup and the number of messages exchanged between applications over the simulation time. The global approach *full* and the approaches applying the communication scheme all reach the optimum, while the local approaches converge to a sub-optimal core allocation. At time step 300, 20 cores are switched off. This decision may be triggered by a run-time management mechanism for temperature or power saving reasons. The figure shows that all approaches react by trying to optimize the speedup of the running applications for the new system constellation. Again, only the global approach *full* and the approaches applying the communication scheme are able to reach the optimal core allocation. However, it can also be seen that approach *full* has a high overhead regarding the number of exchanged messages.

This case study illustrates that (a) run-time management is required to dynamically and optimally react on changes of the system, and (b) sophisticated strategies have to be applied to reduce the management overhead.

B. Scalability

For being able to compare the scalability of the different decentralized core allocation schemes, two sets of experiments are performed. Fig. 5 presents the results of the first set, where 25 applications are concurrently started on a 100 processor many-core. Different scenarios are generated, where 100%, 60%, 50%, 40%, 20%, and 0% of the 25 applications are embarrassingly parallel (with speedup characteristics chosen according to signal/image processing applications), and the rest is moderately parallelizable. For each scenario, 100 test cases are generated, and their average results with standard deviations are depicted in Fig. 5. It shows the resulting avg. speedups (normalized, so that 100% represents the optimal result), as well as the number of messages exchanged between two applications and the convergence time (in number of simulation steps) until the equilibrium is reached.

When having a high amount of embarrassingly parallel applications, optimal or near-optimal speedups can be produced by each approach. However, when other applications are included, local allocation schemes degenerate while the proposed communication scheme guarantees optimality. Finally, if no embarrassingly parallel applications are included, the architecture provides sufficient cores to speedup all 25 applications, and all approaches produce (near-)optimal results.

We furthermore see that in the most cases the communication scheme applied on meshes and chains exchanges less messages than the global allocation scheme. Note that, in addition, in a fully connected approach, the messages have to be sent over longer communication distances, e.g.,

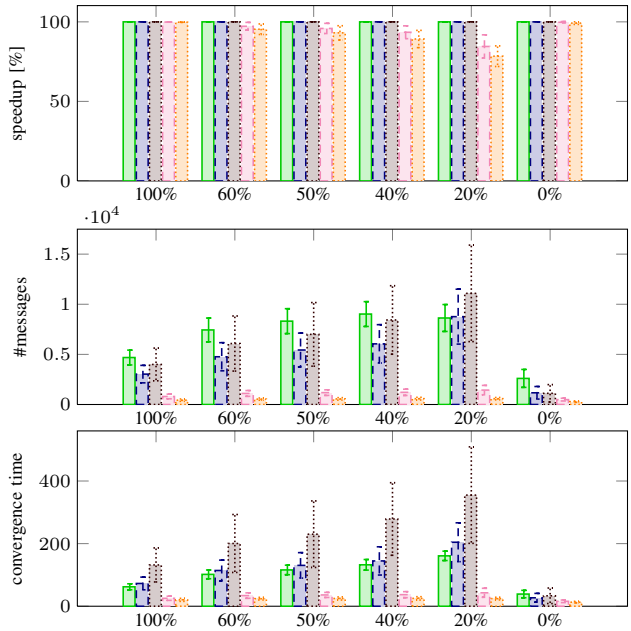


Fig. 5. Results for 100 cores and 25 applications for scenarios which include different amounts of embarrassingly parallel applications (from left to right for each scenario: *full*, *mesh+comm*, *chain+comm*, *local mesh*, *local chain*).

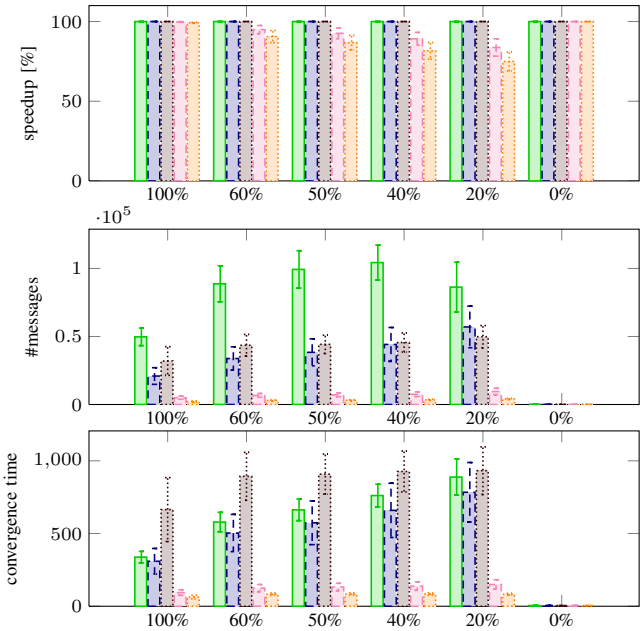


Fig. 6. Results for 500 cores and 40 applications for scenarios include different amounts of embarrassingly parallel applications (from left to right for each scenario: *full*, *mesh+comm*, *chain+comm*, *local mesh*, *local chain*).

on a network-on-chip, resulting in longer latencies. Whereas, neighboring applications in mesh and chain topologies are located adjacent on the chip, and thus have shorter communication distances. While the communication overhead of a fully connected allocation scheme is consequently higher, it only converges slightly faster to the optimal allocation than the communication scheme on the mesh topology. The chain topology requires the longest time.

In the second set of experiments, 40 applications are started on a 500 processor many-core. The results are shown in Fig. 6. The resulting speedup characteristics are similar to those before. However, we see that the mesh topology can reduce the overhead in terms of exchanged messages and convergence time compared to the fully connected and the chain topologies. Comparing the mesh topology with the fully connected approach, an average reduction of 61.4% of the communication and of 13.4% of the convergence time can be observed for the scenario containing 50% embarrassingly parallel applications.

The experiments show that, particularly when we have a mixture of embarrassingly and moderately parallel applications, it becomes necessary to apply either the global allocation scheme or the communication scheme to achieve good results. Comparing these approaches, the overhead scales best for the communication scheme applied on the mesh topology. The local allocation schemes have the lowest overhead, however, to the expense of producing sub-optimal results.

VII. Conclusion

In this paper, we analyzed the optimality of decentralized core allocation for many-core systems, where the goal is to maximize the average speedup of running applications at any point in time. The theoretical results show that basic primitives for claiming and releasing cores suffice so that applications can autonomously reach global optimality assuming global knowledge. In addition, we presented a more realistic model that relies only on limited local communication for negotiation of cores, but still provably reaches a globally optimal core allocation. The experimental evaluation showed that decentralization schemes are of particular importance when a mixture of embarrassingly parallel and other applications is executed. The proposed approach works particularly well for large systems, thus fulfilling the needs for future many-core architectures. For future work, it is necessary to evaluate in how far our theoretical model has to be adapted to more realistic architectural properties. This includes, e.g., communication overheads, memory accesses, and heterogeneous processing cores.

Acknowledgement

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs Journal*, no. 3, pp. 202–210.

[2] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of Design Automation Conference (DAC)*, 2007, pp. 746–749.

[3] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[4] M. Al Faruque, R. Krist, and J. Henkel, "ADAM: Run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of Design Automation Conference (DAC)*, 2008, pp. 760–765.

[5] 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 CODES+ISSS*, 2011, pp. 119–128.

[6] M. Shafique, L. Bauer, W. Ahmed, and J. Henkel, "Minority-game-based resource allocation for run-time reconfigurable multi-core processors," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, 2011, pp. 1 –6.

[7] K. Lam and H. Leung, "An adaptive strategy for resource allocation modeled as minority game," in *Proceedings of SASO*, 2007, pp. 193 –204.

[8] A. B. Downey, "A model for speedup of parallel programs," Berkeley, CA, USA, Tech. Rep., 1997.

[9] P. Sanders and J. Speck, "Energy efficient frequency scaling and scheduling for malleable tasks," in *Euro-Par 2012*, ser. LNCS. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 167–178.