Task Migration for S-Net/LPEL

Stefan Kok, Merijn Verstraaten, Raphael Poss, Clemens Grelck
Informatics Institute, University of Amsterdam, Netherlands
s.kok@student.uva.nl, m.e.verstraaten@uva.nl, r.poss@uva.nl, c.grelck@uva.nl

Abstract

We propose an extension to S-NET's light-weight parallel execution layer (LPEL): dynamic migration of tasks between cores for improved load balancing and higher throughput of S-NET streaming networks. We sketch out the necessary implementation steps and empirically analyse the impact of task migration on a variety of S-NET applications.

1. Introduction

S-NET is a dataflow coordination language and component technology [4, 6]. As a pure coordination language S-NET provides (almost) no means to describe computations of any kind, but it turns regular functions/procedures implemented in a conventional programming language into asynchronously executing, state-less components, named *boxes*. In principle, any conventional programming language can be used, but for the time being we provide interface implementations for the functional array language SAC [5] and for a subset of ANSI C.

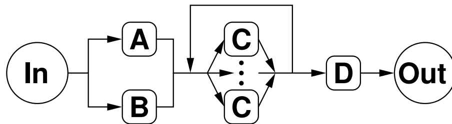

Figure 1. S-NET streaming network of asynchronous components

S-NET components are connected by and solely communicate via uni-directional typed streams. Fig. 1 shows an intuitive example of an S-NET streaming network. Data objects enter the streaming network via a dedicated input component and then travel alongside the streams to compute components. Whenever a data object arrives at box, it triggers a computation as specified by the corresponding box language function (or procedure). During this computation a number of data items may be sent to the output stream to trigger further computations in subsequent boxes. Eventually, data objects reach the dedicated output box, which writes them to file or some other output medium.

S-NET streaming networks are not static, but evolve over time. In Fig. 1 this can be seen best with the box named C. This box is *replicated in parallel* meaning that data objects are routed to some instance of C as indicated by a named index in the data object itself. Hence, instances of C (which could also recursively be complete S-NET networks again) are instantiated as needed. The other dynamic network aspect is *serial replication*. In Fig. 1 this is indicated as a feedback loop around the parallel replication of box C, but in fact there is no feedback in S-NET, only feed forward (among others to rule out deadlock by construction). Effectively, the entire network within the "feedback loop" is dynamically replicated and the replicas are connected by streams one after the other. Data objects entering a serial replication network are routed through an a-priori unknown number of replicas. Before and in between any two such replicas a certain program-dependent condition is checked and the data either routed to the next instance of the replication or to the subsequent network (i.e. box D in the example of Fig. 1).

Serial and parallel replication can arbitrarily be nested, contributing much to the expressive power of S-NET. Consequently, the number of box instances in a running S-NET streaming network quickly grows and demands a smart mapping to compute resources, e.g. the various cores of contemporary server system or cluster node. While the deployment and operational execution of streaming networks is handled by the S-NET runtime system [3], the mapping of boxes to cores as well as the stream communication with suspension and activation of boxes is handled by the underlying *Light-Weight Parallel Execution Layer (LPEL)* [10].

Whenever the S-NET runtime system (due to replication) instantiates a new component, the LPEL layer maps it to some core for execution according to some heuristics. Once mapped a component remains tied to that core for the duration of program execution. This may lead to load imbalances where some cores have a pile of data objects to be processed while others remain idle. The highly dynamic nature of S-NET and the coordination approach that deliberately limits information exchange between compute and coordination layer (Boxes are effectively black boxes) very much limit any form of static analysis and scheduling.

Hence, in the work presented in this paper we extend the LPEL threading layer by means for dynamic task migration. Firstly, we redefine the interface between LPEL and the S-NET runtime system box language interface to temporarily yield control to LPEL between any two data objects to be processed by some box. This gives LPEL a handle to change the mapping of components on this occasion. Secondly, we define an asynchronous scheduler task (a migration controller) that continuously observes the load balancing status of a running streaming network. According to selectable heuristics the migration controller may choose to asynchronously update the mapping of components to cores. The LPEL layer in turn implements the re-mapping, which becomes effective with the next data object to be processed.

The remainder of the paper is organized as follows. In Section 2 we provide additional background information on S-NET, its runtime system and the LPEL threading layer. Section 3 describes our technical contribution on task migration in greater detail, followed by an experimental analysis in Section 4. In Section 5 we draw conclusions and outline directions of future work.

Copyright is held by the author/owner(s).
FD-COMA 2013 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, Berlin, Germany, January 21-23, 2013
ACM SIGPLAN conference style.

2. S-NET: Design and Implementation

2.1 S-Net language

The basic building blocks of S-NET streaming networks are boxes. Each box is connected to the rest of the network by two typed streams: one for input and one for output. Following the data flow principle, a box is triggered by receiving a record on its input stream, upon which the box applies its *box function* to the incoming data object. As pointed out before, this box function is implemented in a *box language* selected for suitability in the relevant application domain. During execution the box may send records to its output stream. As soon as execution of the box function has finished, the box is ready to receive and process the next item on the input stream.

It is a distinguishing feature of S-NET that it neither introduces streams as explicit objects nor defines network connectivity by explicit wiring. Instead, S-NET uses algebraic formulae for describing streaming networks in a much more abstract way. The restriction of the boxes to single input streams and single output streams (named the *SISO principle*) is essential for this. S-NET provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property: any network, regardless of its complexity, again is an SISO entity.

Let `A` and `B` denote two S-NET networks or boxes. Serial composition (`A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. Parallel composition (`A|B`) constructs a network where incoming records are either routed to `A` or to `B`; their output streams are merged to form the compound output stream. The type system controls the flow of records. Serial replication `A*type` constructs an infinite chain of replicas of box or network `A` connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. Last not least, parallel replication `A!<tag>` also replicates box or network `A`, but this time the replicas are connected in parallel. All incoming records must carry a property `<tag>` whose integer value determines the replica to which the record is routed. These four orthogonal network construction principles are sufficient to define complex streaming networks.

For more detailed information on the S-NET language we refer the interested reader to [4, 7]

2.2 S-Net runtime system

The S-NET runtime system [3] is responsible for deployment and operation of streaming networks. Thanks to the serial and parallel replication combinators networks evolve dynamically, and thus deployment and operation are not two distinct phases, but rather alternating, i.e. the operation of some network component may trigger another replication and, thus, the further deployment of network structures.

Furthermore, the S-NET runtime system turns implicit split in merge points in the construction of networks into active internal components that explicitly split an incoming stream into two (or more) outgoing streams by implementing the routing protocol or that merge two (or more) incoming streams into a single outgoing streams. As internal routing components these splitters and mergers do not comply to the SISO principle, but effectively implement the various routing protocols derived from the S-NET network combinators. Fig. 2 illustrates a partially deployed state of the example network introduced in Fig. 1. For illustration reasons, splitters and mergers are represented as (anonymous) triangles, but in fact each split and merge component does have a proper identity.

Each component, both internal split and merge components as well as user-level boxes, runs a simple event loop. First, a component checks the input stream for data. If the input stream is empty the component suspends. Otherwise, the first data item on the input stream is consumed and processed. If this processing requires sending a data item to an output stream, the component may suspend on a full output stream. If a component completes processing one item, it continues from scratch. Taking a data item out of a stream automatically wakes up components suspended on sending data to this stream. Likewise, adding a data item to some stream wakes up components suspended on reading from this stream.

2.3 LPEL threading layer

The S-NET runtime system relies on basic threading mechanisms such as task creation, suspension, wake-up and termination. Such mechanisms are essentially provided by any multithreading library, including PThreads to name a specific one. However, even fairly simple S-NET streaming networks with nested replication combinators induce a large number of components to be instantiated at runtime. This motivates a two-layered approach where a small number of kernel threads essentially abstract the compute resources (cores) to be used while the tasks demanded by the S-NET runtime system are implemented by light-weight user-level thread contexts that are cooperatively scheduled among the kernel threads.

The Light-Weight Parallel Execution Layer (LPEL) [10] is such a two-level threading implementation tailored to the needs of the S-NET runtime system. On initialization LPEL creates a user-specified number of *worker* threads. These workers are kernel threads and, thus, preemptively scheduled by the operating system to the available cores. The general assumption is that the number of workers does not exceed the number of cores, and workers are bound to individual cores to effectively deactivate the operating system scheduler.

The instantiation of some S-NET component during a deployment phase incurs the creation of an LPEL *task*, or light-weight thread. This task is assigned to some worker based on some heuristic. Important for the subject of this paper: tasks are never reassigned (or migrated) from worker to another once created. Each worker has a priority queue of ready tasks and a queue of suspended tasks that wait for data on an empty stream or for space on a full stream. Reading from and writing to streams accordingly moves tasks between these queues not dissimilar to standard operating system procedures.

3. Task Migration

In this section we will discuss the task migration framework developed for S-NET and LPEL.

3.1 Challenges

Conceptually, S-NET boxes are nothing but (pure) functions that are called on some incoming data item. As a result, migrating tasks between workers should be as simple as sending the input data to a different worker and having the next function invocation performed by that worker. However, as already pointed out in Section 2.2 the S-NET runtime system implements boxes as long-lived tasks with an internal event loop triggered by receiving data on th input stream and by sending data to the output stream.

Migration of such long-lived tasks would involve halting the task, migrating the task's current state (including state of the computation, such as the stack) and then unpausing the task. This would be doable in a shared memory system, but with an eye on DISTRIBUTED S-NET [2] and NUMA architectures, we want to make the migration of state as explicit as possible to simplify carrying over our current work to these settings in the future.

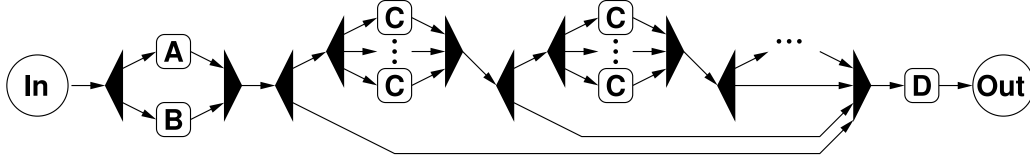

Figure 2. Runtime representation of the S-NET streaming network of Fig. 1

Another migration challenge is that any overhead introduced by a migration mechanism and its associated heuristics should be less than the performance gained by performing the migration, otherwise there is nothing to be gained from migrating tasks.

3.2 Respawning

As a first step towards task migration we modified the S-NET runtime system to expose more fine-grained concurrency: each task becomes a one-shot activation of an entity that handles a single input record. The simple implementation of this idea is to have every S-NET task spawn a new copy of itself upon termination. However, this would introduce a significant amount of overhead for the common case where a task does not migrate. This is due to LPEL having to do some expensive allocation upon LPEL thread creation (such as the task's stack) which can be reused if the task does not migrate to a different worker.

To solve this issue we implemented a continuation option in LPEL where each thread has an optional continuation associated with it. If this continuation is set, LPEL will run it in the thread's context, as soon as the previous execution finishes. The S-NET implementation of spawning the next activation can then be achieved by setting the continuation to the current function.

The result is that after every activation of an S-NET entity, control flow returns to the LPEL layer to start the task's continuation. At this point LPEL is in a position to check whether the task should be migrated to another worker. If the task has to be migrated the LPEL code can spawn a new thread on a different worker to execute the continuation.

3.3 Synchronous vs asynchronous migration

Now that LPEL has gained the technical capability to migrate tasks between worker, we have to think on how we decide when to migrate a task and when not. An approach that immediately comes to mind is to define a placement oracle and on each continuation consult that oracle. This would be fairly simple, but would likewise introduce a significant amount of overhead as soon as the oracle requires a non-trivial amount of computation because every worker has to do a blocking invocation of the placement oracle upon each continuation of an S-NET task.

Rather than following the above synchronous approach, we decided to make placement decisions asynchronously from task processing. For this we extend the LPEL thread control structuer with a *next-worker* field that indicates the worker on which the next invocation should run. This means that LPEL checks whether the current and next workers are the same, if so the continuation is invoked. If, however, the next worker is different from the current, LPEL spawns the continuation on the new worker and thus effectively migrates the task.

3.4 Placement scheduler

The open question is still where, when and how the *next-worker* field is updated. As a starting point we introduce the notion of a placement scheduler. This is a conceptual task in the LPEL system that periodically inspects tasks and determines whether they should migrate on their next invocation. The placement scheduler is set up so that it can use any arbitrary oracle to decide the new placement. As a small starting experiment to test the migration code and placement scheduler we implemented two very simple strategies for placement. To accommodate these strategies we added optional hooks to each scheduling event. These hooks update any strategy specific state that is used by the placement scheduler to determine placements.

3.5 Placement strategies

The first implemented strategy is random migration. After every invocation a task is marked for migration with probability p. The placement scheduler updates the next worker field of selected tasks with a random worker. This strategy can then be used as a baseline to see whether placement has any effect (positive or negative) at all, in terms of performance gain or overhead introduced.

The second strategy does placement based on the waiting times of tasks. That is, the time that a task is runnable, but not running. The waiting time T_{ready} is the sliding window average of the past n run-suspend cycles. For every worker we maintain the average $\mu_{T_{ready}}$ of the T_{ready} of each task on that worker. A task is selected for migration if its T_{ready} is larger than the $\mu_{T_{ready}}$ of its worker. The task is then migrated to the worker with the lowest $\mu_{T_{ready}}$. The goal of this strategy is to minimize the time a ready task spends waiting to run, aiming at increasing the average utilization of workers and balancing their loads.

4. Analysis

In this section we investigate the performance impact of task migration. Our first implementation attempt did not use scheduling hooks to update the migration state of tasks. We benchmarked[1] this implementation using a domain-decomposition implementation of raytracing [9]. The results of these benchmarks quickly showed that the placement scheduler's locking eliminated any scaling S-NET had, as shown in Figure 3 and Figure 4.

After these disappointing results we redesigned the implementation of the placement scheduler to avoid unnecessary locking and use atomic operations where synchronization between threads could not be avoided. This new implementation reduced the overhead created by the placement scheduler to an insignificant amount, as illustrated by the graphs in Figure 5 and Figure 6. The scaling is roughly identical between the runs with placement and those without.

However, our goal was to improve the performance of S-NET applications, which is not occurring in these benchmarks. The raytracer, with its domain-decomposition implementation is not ideal for testing our placement scheduler, as its workload is very static. We proceeded to explore several other example S-NET applications with various threshold to examine the performance impact of placement on other workloads.

We selected three benchmarks with very different workloads. Two of the applications were used as benchmarks in previous research, an ant colony optimization program[1] and acoustic target

[1] All experiments are done on a shared memory machine, a 12 core Intel(R) L5640 2.27 GHz Xeon(R) CPU, with 24 Gigabytes of RAM.

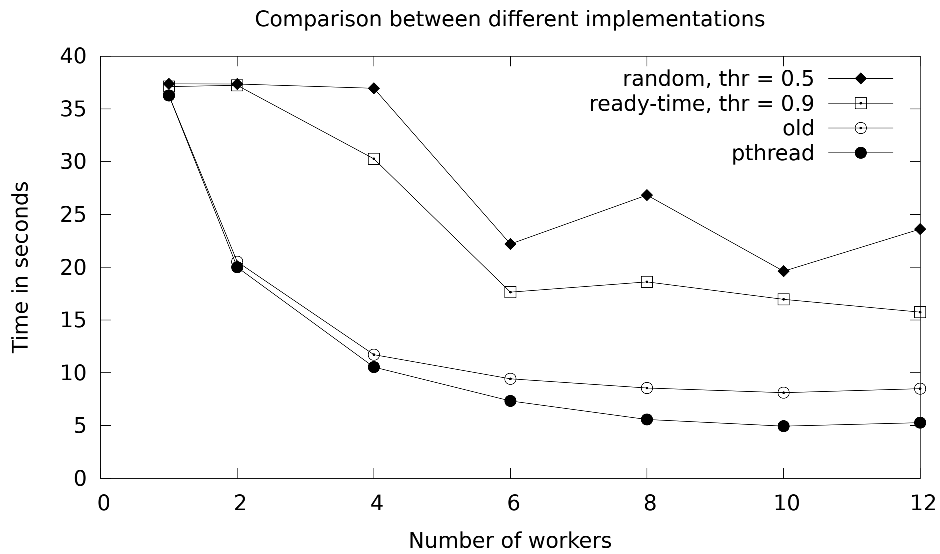

Figure 3. Raytracing durations for different placement strategies.

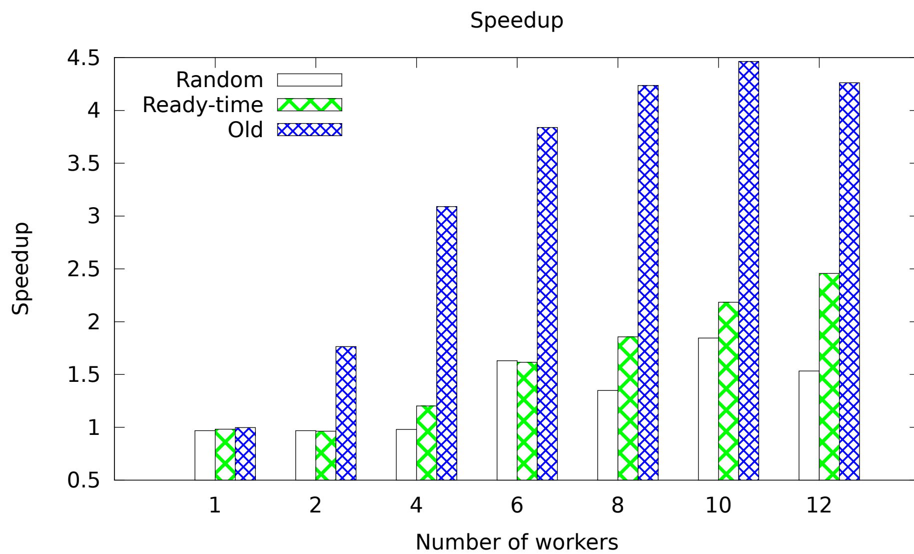

Figure 4. Raytracer scaling for different placement strategies.

FIXME

Figure 5. Benchmark runtimes for different placement strategies.

FIXME

Figure 6. Benchmark scaling for different placement strategies.

tracker using the MTI-STAP algorithm[8]. In addition to these we used an example network generated by our automatic benchmark generator. The results of these benchmarks are shown in Figure 7, Figure 8 and Figure 9.

As shown in these graphs the placement

5. Conclusion

Acknowledgements

The work has been funded by the EU FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, project no. 248828).

References

[1] W. Cheng, F. Penczek, C. Grelck, R. Kirner, B. Scheuermann, and A. Shafarenko. Modeling streams-based variants of ant colony optimisation for parallel systems. In *HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'12), Paris, France*, pages 11–18, 2012.

[2] C. Grelck, J. Julku, and F. Penczek. Distributed s-net: Cluster and grid computing without the hassle. In *Cluster, Cloud and Grid Computing (CCGrid'12), 12th IEEE/ACM International Conference Ottawa, Canada*. IEEE Computer Society, 2012. to appear.

[3] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer-Verlag, 2011.

[4] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

[5] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[6] C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[7] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[8] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrière, and E. Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079 – 2088, 2010. ICCS 2010.

[9] F. Penczek, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, N. Bagherzadeh, and C. Grelck. Message Driven Programming with S-Net: Methodology and Performance. *Parallel Processing Workshops, International Conference on, San Diego, USA*, 0:405–412, 2010.

[10] D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technical University of Vienna, Vienna, Austria, 2011.

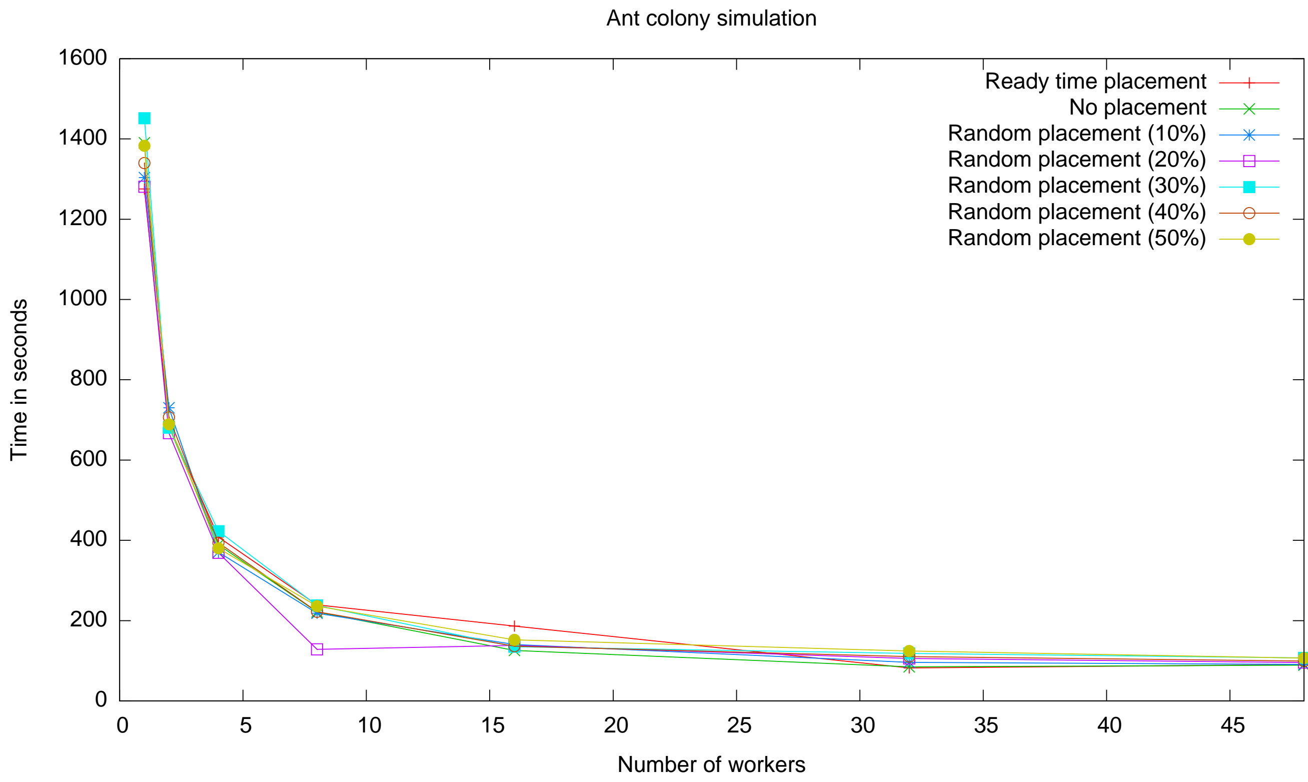

Figure 7. Ant colony results.

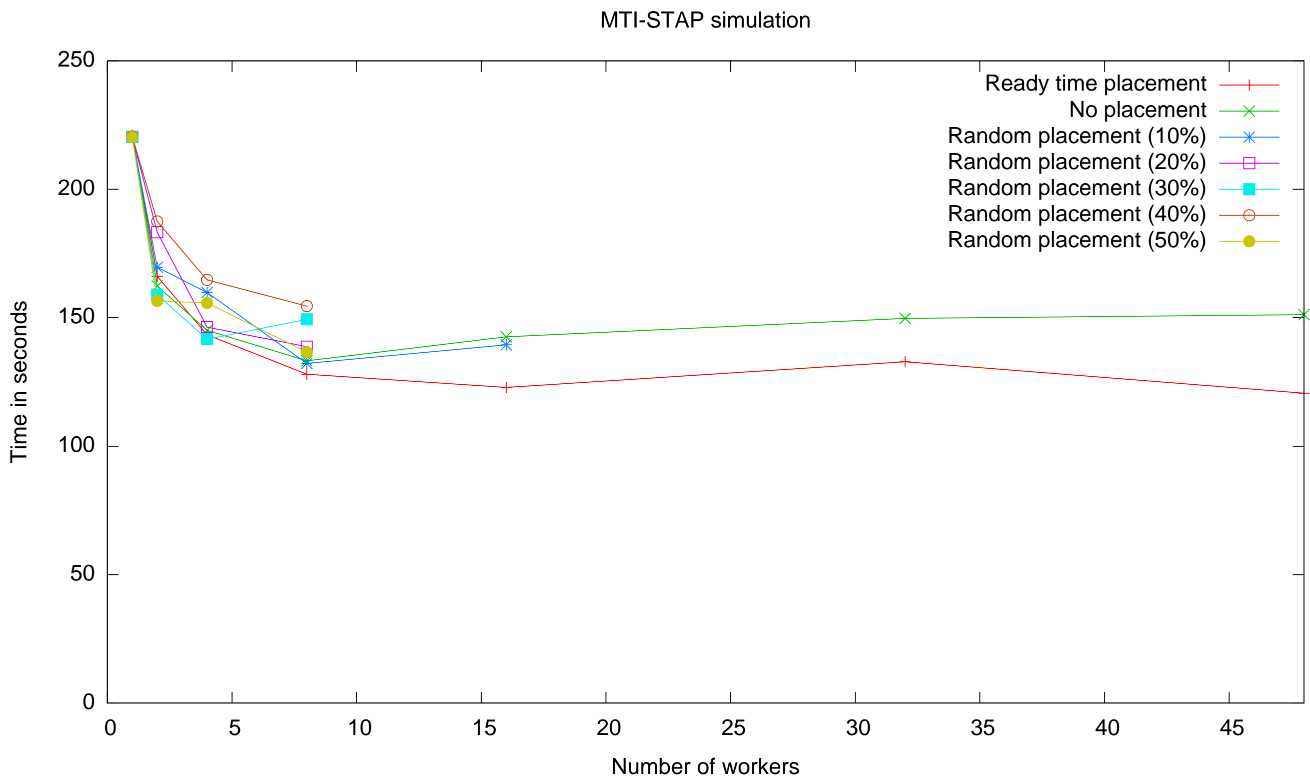

Figure 8. MTI-STAP results.

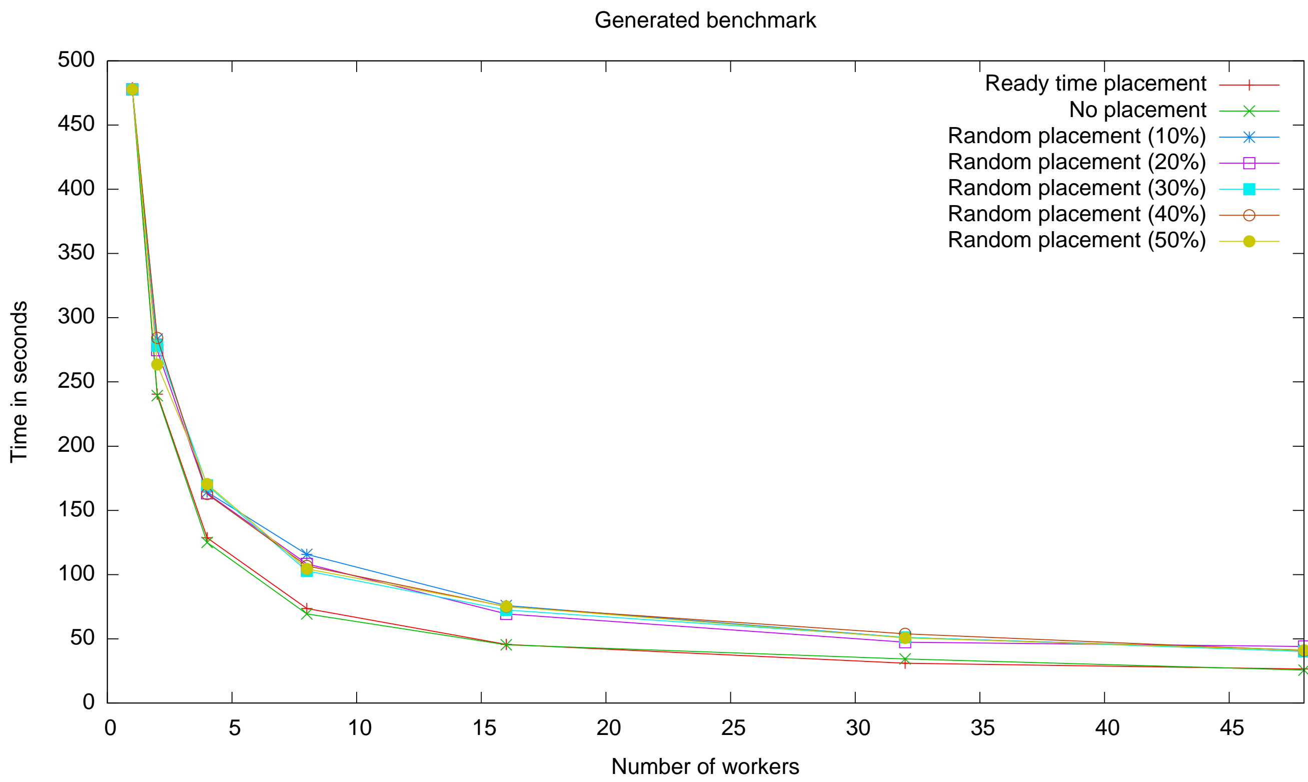

Figure 9. Generated benchmark results.