Industrial and
Systems Engineering

Could We Use a Million Cores to Solve an Integer Program?

Thorsten Koch

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

Ted Ralphs

Lehigh University, Bethlehem, PA 18045 USA

Yuji Shinano

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

COR@L Technical Report 12T-019

LEHIGH
UNIVERSITY

COR@L
COMPUTATIONAL OPTIMIZATION
RESEARCH AT LEHIGH

Could We Use a Million Cores to Solve an Integer Program?

Thorsten Koch*[1], Ted Ralphs†[2], and Yuji Shinano‡[3]

[1]Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
[2]Lehigh University, Bethlehem, PA 18045 USA
[3]Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

June 24, 2012

Abstract

Given the steady increase in cores per CPU, it is only a matter of time before supercomputers will have a million or more cores. In this article, we investigate the opportunities and challenges that will arise when trying to utilize this vast computing power to solve a single integer linear optimization problem. We also raise the question of whether best practices in sequential solution of ILPs will be effective in massively parallel environments.

1 Introduction

Prediction is very difficult, especially about the future.
Niels Bohr

Until about 2006, one could rely on the fact that advances in solver algorithms over time would be augmented by inevitable increases in the speed of the computing cores of central processing units (CPUs). This phenomenon led to two decades of impressive advances in solvers for linear and integer optimization problems [12, 31]. Since then, the single-thread performance of processors has increased only moderately and what meager improvements have occurred are mainly due to improvements in CPUs, such as better instruction processing and larger memory caches. The raw clock speed of general purpose CPUs has stayed more or less constant, topping out at 5 GHz with the IBM Power6 CPU. The increases in clock speed that occurred so reliably for decades have now been replaced by similar increases in the number of processing cores per CPU. Figure 1 summarizes CPU development since 1985. Currently, CPUs with 12 cores are available from AMD and Intel is planning to release specialized CPUs with 50 cores for high-performance computing next year.

Not surprisingly, current trends in supercomputing are focused around the use of ever-increasing numbers of computing cores to achieve increases in overall computing power. Today, the ten fastest machines in the world (as measured by the Linpack benchmark) have 180,000 cores on average and it is likely that a million cores will be available soon. While increased clock speeds contribute

*koch@zib.de

†ted@lehigh.edu

‡shinano@zib.de

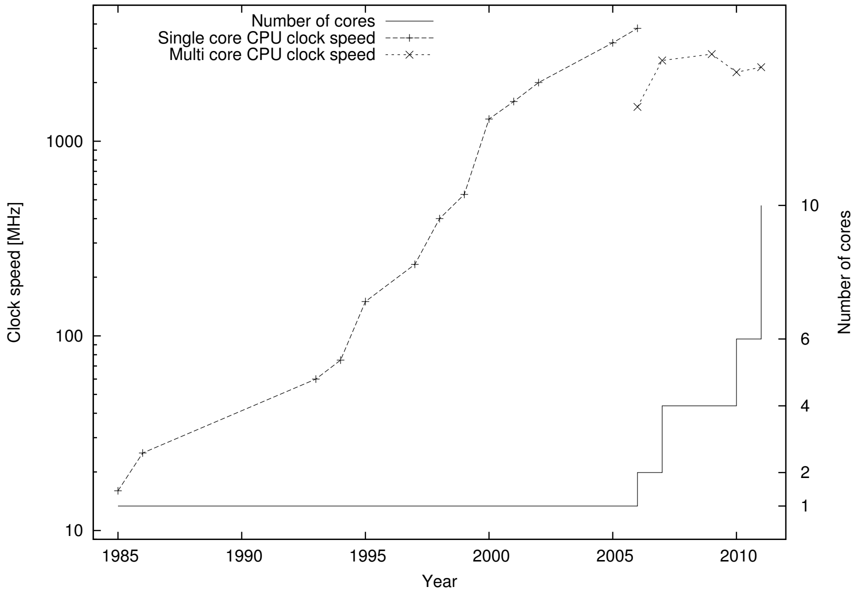

Figure 1: Clock speed and number of cores for Intel processors from the 80386DX in 1985 to the Westmere-EX in 2011

directly to increases in the performance of a solver over time, it is not as clear whether increases in the number of processing cores will be equally beneficial. In general, the challenge of effectively utilizing such complex architectures lies both in dividing the computation into tasks of roughly equal difficulty and in ensuring that the data required to perform each of these task is locally available to the core assigned to perform it. This becomes increasingly challenging as the number of available cores gets larger, especially if the set of tasks is essentially unknown at the outset of the computation.

The natural question that arises is whether and how we can harness such vast computing power to solve difficult optimization problems. In what follows, we address this question with respect to integer linear optimization problems (ILPs), which are optimization problems of the form

$$\min\{c^\top x \mid Ax \leq b, x \in \mathbb{Z}^n\}$$

where $A \in \mathbb{Q}^{m\times n}$, $b \in \mathbb{Q}^m$, and $c \in \mathbb{Q}^n$. Nevertheless, the discussion applies similarly to the solution of a wide range of problem classes for which enumeration algorithms are generally employed.

Why is it important to think now about what we can do with a million cores? After all, such systems are not yet available and may even seem fanciful by today's standards. However, as we argue below, it is not unreasonable to expect million core systems in the relatively near future. Currently, a major milestone anticipated in the development of supercomputers is the development of a so-called Exascale machine. By any forecast, such a machine will employ a million cores or more. A major point of concern regarding future supercomputers is the amount of energy they will consume. The current number one supercomputer, the *K computer*, requires nearly 10 MW, even though it is considered comparatively energy efficient. Since the amount of energy a CPU needs grows superlinearly with clock speed, one way of reducing power consumption is to trade clock speed for cores, i.e., have more but slower cores. This will amplify the problems we envision in this paper regarding the scaling of ILP algorithms.

As it is generally the case that the power of a supercomputer today will become available in a desktop system within a few years, we may see such computing power and many-core designs on the desktop within a decade. Even if all this were not to come to pass, there are important applications, e.g., infrastructure planning for power, water, and gas networks, where the question is not whether we can solve the associated models at a reasonable cost with current desktop computing technology, but rather whether we can solve them by any means at our disposal.

No matter what the specifics of future computing platforms turn out to be, it is clear that parallel algorithms able to run effectively in massively parallel environments will have to be developed if we expect to continue to see advances in the size and difficulty of the ILPs we can solve. It is therefore important to question what will be different in such an environment. Do the assumptions we normally make about the way an ILP should best be solved hold? It is very likely that this is not the case. We therefore endeavor to undertake this thought experiment as a means to examine what long-held beliefs we might have to abandon in order to achieve efficiency in massively parallel computing environments.

It must be emphasized that what follows is highly speculative and should be viewed with appropriate skepticism. In order to answer the question posed in the title, we make assumptions about the future of both computing and optimization technologies based on our best current understanding of the trends. Performance numbers given in the paper are based on a few preliminary experiments that the authors believe are useful to explain certain trends. These experiments should, however, be questioned, reproduced, and extended before drawing further conclusions.

With respect to solution technologies, we address algorithms that are enumeration based, i.e., explore a search tree via a traditional LP-based *branch-and-bound* (B&B) algorithm. Many of the ideas we explore here apply to the more broadly defined class of tree search algorithms, which in turn fall into the even broader class of algorithms that take a *divide-and-conquer* approach, recursively dividing the problem domain to obtain subproblems that can be solved (more or less) independently of each other. Although such algorithms appear to be “embarrassingly parallel,” this appearance is deceiving because the question of how to effectively divide the search space is provably difficult in itself (see [35]) and must be answered as the solution process evolves. We primarily address algorithms that derive bounds on subproblems by solving relaxations of the original ILP that are (continuous) linear optimization problems (LPs) obtained by dropping the integrality requirements on the variables. These relaxations are typically augmented by dynamically generated valid inequalities (see, e.g., [48, 2] for details regarding general ILP solving and [51, 47, 6, 43] for distributed memory solution techniques).

The above assumptions and the setting we consider mean that the processing of individual subproblems in the search tree is non-trivial. The complexity of these computations as atomic units has been growing steadily over time as more sophisticated (and time-consuming) methods of computing bounds are developed. Investigation of so-called *branch-and-cut* algorithms (see, e.g., [24, 41]) has lead to a tremendous reduction in the size of the search tree needed to solve a given instance, leading to substantial reductions in the time required to solve instances on a *sequential* computing platform. In fact, it is now often possible to solve ILPs without any enumeration at all. Unfortunately, although this leads to big improvements in computation time on sequential computing platforms, it confounds efforts to parallelize computations in the most straightforward way on parallel computing platforms. All of this makes it necessary to consider methods of parallelizing the computations undertaken in processing an individual node in the search tree, which is a challenging problem in its own right. In Section 7, we address methods for parallelizing the solution of the LP relaxation, which seems the most promising approach to this at present.

Our main purpose in making these assumptions is not to lend credence to the de facto direction in which efforts to parallelize solution algorithms have so far gone, but to evaluate whether this direction is sustainable and if parallelized versions of our current state-of-the-art algorithms will scale to a machine with a million cores. We will not investigate the question of whether a different methodology for solving ILPs should be used. So far, despite considerable research, there is no promising alternative algorithm known to the authors. There are many ideas one could try, e.g., basis reduction [18] or primal methods [1], but it appears unlikely any of these will be an improvement in the general case, though it is entirely possible that some alternative that is more effective in a parallel setting will eventually emerge. One direction that seems promising, but that we do not consider here, is the use of decomposition algorithms for parallelization of the bounding operation of individual search tree nodes, such as in [22]. These have been shown to be highly effective for certain specialized applications.

With respect to computing technologies, we assume that the trends observed in the last five years will continue. For the purposes of describing a future million-core system, the word *core* will be used to denote the part of a CPU that executes single sequences of instructions and the word *thread* to denote a *sequential program* consisting of such a set of instructions. A *parallel program* will be taken to consist of a set of threads executing simultaneously either (1) on multiple cores of the same CPU, (2) on multiple cores of different CPUs sharing a memory space (*shared memory execution*), or (3) on multiple cores of different CPUs that may or may not share memory (*distributed memory execution*). We assume that each core executes a single thread, so that there are never more threads than cores (though there might be more cores than threads). For the foreseeable future, we assume that computers having a million cores will necessarily use distributed memory with the overall system consisting of clusters of *processing elements* (PE). We define a PE loosely as one shared memory node within this distributed system. One PE might have one or more CPUs, with each CPU having several cores. More details on what such a system might look like are given in Section 3.

2 Solvability of ILPs

Before we address the question of which ILPs might be tackled effectively in parallel with multiple cores, we first examine the reasons why ILPs cannot be solved by today's solvers using a sequential algorithm (see also [13]).

1. **Weak formulation**. Because the solution space of an ILP is defined implicitly as the integer points inside a given polyhedron, the same ILP can have a wide variety of alternative formulations. The specific formulation chosen has a strong influence on the difficulty of solving the problem. The "strength" of the formulation is usually measured by how close the bound obtained from solving the LP relaxation is to the optimal solution value. If the bound obtained from the LP relaxation is far from the optimum value of the ILP, it is likely that a large enumeration tree will be needed to solve the problem. Another reason why a particular formulation may be ineffective is the presence of symmetry in the solution space, i.e., many equivalent solutions of similar cost, again requiring substantial amounts of enumeration [9, 37].

   Weak models can arise due to "improper" modeling on the part of the user, but there are cases in which no better model is known. In some such cases, the solver may actually be capable of automatically reformulating, but often, an improved starting formulation is necessary to overcome this difficulty. Note that just because a formulation produces poor bounds does not mean the optimal solution will be difficult to find. In many cases, the optimal solution can be easily produced—it is proving optimality that is difficult (see Section 6.1). Empirically, it can be often observed that in case of weak formulations the branch-and-bound procedure becomes quite ineffective, producing search trees large enough the addition of more computing resources has little effect.

2. **Poor numerical conditioning**. Poor numerical conditioning means that because of the specific structure of the matrix describing the instance, an accurate solution to the LP relaxation may not be easily obtained, given the limited numerical precision of current CPUs. Typically, this must be addressed with either an improved model or a more robust solution technique. For instance, one might use a so-called "exact" solver employing rational arithmetic [17]. An easier-to-implement alternative would be to use additional branching to increase precision at the expense of increasing the size of the search tree (see item 4). It is also possible to improve the situation (possibly with hardware) by the use of quad-precision floating point arithmetic, but this will also increase the computational burden (see item 5).

3. **Difficult-to-find primal solutions**. In some cases, the structure of a particular problem makes it difficult even to find a *feasible* solution, let alone an optimal one. This can arise simply because very few such solutions exist, i.e., the effective feasible region is small (and unknown). Simply through brute force enumeration, we anticipate that more cores are likely to be helpful in this case. Nevertheless, there is some empirical evidence that failure to find the primal solution is seldom the reason for not being able to solve an ILP. In contrast to the case of weak formulation, finding a dual bound equal to the optimal value may be easy in this case (see Section 6.1), though solving the problem is not.

4. **Large enumeration trees**. Any of the above phenomena could contribute to the generation of large enumeration trees that would simply take too long to explore sequentially. In such cases, the ability to evaluate more B&B nodes clearly helps, as long as the tree is balanced enough to effectively divide the computation.

5. **Long node processing times**. Typically, this situation arises either because solving the initial LP relaxation is difficult or because reoptimization (solving subsequent LP relaxations from a warm start) is not efficient. It may also arise due to excessive time spent generating valid inequalities or an excessive number of inequalities generated. In the absence of a better way to solve the linear relaxations, we expect that this can be resolved through the employment of additional cores.

6. **Model is too big**. If there is not enough memory to solve the LP relaxation of the model on a single PE, solution is impossible with most modern solvers. We investigate whether a distributed computing approach can overcome this challenge in Section 7.

In this paper, we focus primarily on showing how using more cores may help for items 4, 5, and 6 above, though, as we already noted, the ability to deal with these cases may also be of indirect benefit in other cases.

Table 1: Top500 list as of June 2011

		Build	Total	CPU		Cores
#	Computer	Year	Cores	Family	GHz	/CPU
1	K computer	2011	548,352	Sparc	2,00	8
2	NUDT TH MPP	2010	186,368	EM64T	2,93	6
3	Cray XT5-HE	2009	224,162	x86_64	2,60	6
4	Dawning TC3600 Blade	2010	120,640	EM64T	2,66	6
5	HP ProLiant SL390s	2010	73,278	EM64T	2,93	6
6	Cray XE6	2011	142,272	x86_64	2,40	8
7	SGI Altix ICE 8200EX/8400EX	2011	111,104	EM64T	3,00	4
8	Cray XE6	2010	153,408	x86_64	2,10	12
9	Bullx super-node S6010/S6030	2010	138,368	EM64T	2,26	8
10	BladeCenter QS22/LS21 Cluster	2009	122,400	Power	3,20	9

3 A Million Core System

A supercomputer is a machine for turning a
compute-bound problem into an I/O-bound problem.
Ken Batcher

As we briefly discussed in Section 1, we expect future systems to be composed, loosely speaking, of clusters of PEs. Table 1 shows the top ten systems on the June 2011 Top500 list[1], all of which fit this description of a supercomputer. If we assume the availability of 32 core CPUs in the near future and further assume four CPUs per PE, then each PE will have about 128 cores with a single shared memory space. A cluster consisting of 8,000 of these PEs would have a million cores. Comparing these numbers to the *K computer*, which has eight cores per CPU, four CPUs per PE, and 17,136 PEs, it is easy to argue that once 32 core CPUs are available, systems with one million cores will quickly become standard in supercomputing.

An important overarching trend in the current evolution of computing technology is the ever-increasing complexity of the so-called *memory hierarchy.* The challenge of parallel computing can be viewed at a high level as the challenge of having the data required for computations in the right place at the right time. From the local viewpoint of a single core, the global store of problem data can be divided into classes according to the length of time it will take to access it. Data in the L1 cache attached directly to the core can be accessed quickly, whereas data stored in higher levels of cache, in local memory attached to the core, in local memory attached to other cores, and finally on other PEs can only be accessed at the cost of increasingly higher retrieval times (known as *latency*), as shown in Table 2. As computing devices grow more complex and the memory hierarchy gains more levels, the problem of where to store data for effective computation will continue to grow ever more complex [49].

We envision two basic alternatives with respect to the above basic design for tomorrow's supercomputers. The first is specialty machines like IBM's BlueGene and the second is large aggregations of "commodity" PCs. For specialty machines, the number of cores per PE will likely be much higher than what was described above, while the memory per core will be less. Such machines will also likely have extremely fast interconnect and thus low latency (compressed memory hierarchy). For

[1] www.top500.org

Table 2: Relative Memory Latency, [33, 42]

Data Hierarchy	Normalized Latency
Layer	Access Times
L1 Cache	1×
L2 Cache	4×
L3 Cache	16×
Local / Remote Memory	$50 - 700\times$
Network	>1,000×
Disk	>3,000,000×

aggregations of PCs, one might expect something more like 64 cores per PE, in which case one would need about 16,000 PEs and a massive amount of electric power to get a million cores. These details do not substantially change the broad conclusions drawn in what follows.

The *K computer* has 2 GB of memory per core, while contemporary PCs have anywhere from 2 to 16 GB of memory per core. Assuming 4 GB of memory per core for the million core machine, we would have 512 GB per PE, resulting in four petabytes of memory for the machine in total. It seems reasonable to further assume that, compared to today's computers:

- the total memory of the system will increase;
- the memory per PE will also increase; but
- the memory per core will rather decrease.

We note that with such a system, reliability becomes a serious issue. According to [46], memory errors are correlated with the number of memory modules installed, but not with the number of bits. Nevertheless, the authors write that "Uncorrectable errors of 0.22% per DIMM per year make a crash-tolerant application layer indispensable for large-scale server farms." In practice, it can be observed that the number of failures depends on the number of components, e.g., DIMMs, CPUs, disks, and not on the capacity, e.g., bits, cores, bytes, of those components. Since the increase in capacity is coming primarily from increased integration, e.g., more bits per DIMM, more cores per CPU, more bytes per disk, there is hope that the reliability of these systems will stay somewhat constant over time. Nevertheless, for high-end systems, reliability tends to be a problem.

For the following analysis, we assume that the system is built using general-purpose CPUs. The reason for this assumption is the observation that the properties of CPUs and GPUs are converging over time. CPUs are continuing to get more cores, as evidenced by Intel's announced *Knight's Corner* CPU with approximately 60 general-purpose cores, each with four fold hyper-threading and AVX vector extensions. On the other hand, each new generation of GPUs has allowed a more flexible programming model and has incorporated features previously belonging only to CPUs. Nvida is marketing the Tesla accelerator as the first *General Purpose GPU* and has implemented double precision floating point operations. Finally, regarding the operations that are needed for implementing an ILP solver, no efficient GPU acceleration capability is currently known.

4 Benchmarking

An important question that must be addressed in order to perform meaningful experiments with large parallel systems is how to measure performance. Even in determining a proper experimental set-up, there are a number of challenges to be faced, as we detail below.

4.1 Traditional Measures of Performance

Generally speaking, the question to be answered with respect to a parallel algorithm running on a given parallel system is whether it "scales," i.e., is able to take advantage of increased resources such as cores and memory. The most commonly used measure of scalability is the *efficiency*, which is an intuitive and simple measure that focuses on the effect of using more cores, assumed to be the bottleneck resource, to perform a fixed computational task (e.g., solve a given optimization problem). The efficiency of a parallel program running on N threads is computed as

$$E_N := (T_0/T_N)/N,$$

with T_0 being the sequential running time and T_N being the parallel running time with N threads. Generally speaking, the efficiency attempts to measure the fraction of work done by the parallel algorithm that could be considered "useful." An algorithm that scales perfectly on a given system would have an efficiency of $E_N = 1$ for all N. A related measure is the *speed-up*, which is simply

$$S_N := NE_N.$$

Reasons for a loss of efficiency as the number of threads is increased can generally be grouped into the following categories.

- *Communication overhead*: Computation time spent sending and receiving information, including time spent inserting information into the send buffer and reading it from the receive buffer. This is to be differentiated from time spent waiting for access to information or for data to be transferred from a remote location.
- *Idle time (ramp-up/ramp-down)*: Time spent waiting for initial tasks to be allocated or waiting for termination at the end of the algorithm. The ramp-up phase includes inherently sequential parts of the algorithm, such as time spent reading in the problem, processing the root node, etc., but also the time until enough B&B nodes are created to utilize all available cores. The ramp-up and ramp-down time is highly influenced by the shape of the search tree. If the tree is "well balanced" and "wide" (versus deep), then ramp-up time will be minimized.
- *Idle time (latency/contention/starvation)*: Time spent waiting for data to be moved from where it is currently stored to where it is needed. This can include time waiting to access local memory due to contention with other threads, time spent waiting for a response from a remote thread either due to inherent latency or because the remote thread is performing other tasks and cannot respond, and even time spent waiting for memory to be allocated to allow for the storage of locally generated data.
- *Performance of redundant work*: Time spent performing work (other than communication overhead) that would not have been performed in the sequential algorithm. This includes the evaluation of nodes that would not have been evaluated with fewer threads. The primary

reason for the occurrence of redundant work is differences in the order in which the search tree nodes are explored. In general, one can expect that the performance of redundant work will increase when parallelizing the computation, since information that would have been used to avoid the enumeration of certain parts of the search space may not yet have been available (locally) at the time the enumeration took place in the parallel algorithm. However, it is entirely possible that the parallel algorithm will explore fewer nodes in some cases.

The degree to which we can control/limit the impact of these sources of overhead determines the efficiency of the algorithm.

4.2 Measures of Performance for Branch and Bound

Although efficiency is a commonly quoted measure of performance, it is not ideal in this setting for a number of reasons. First, it assumes the use of a fixed test set on which the algorithm can be run to completion on a single thread. For a million core system, we do not expect problems that can be solved in a reasonable amount of sequential computing time to be of much interest. It is of course possible to measure efficiency with respect to a different baseline number of threads, but even this may not be practical with a million core system where running times may be limited. A second difficulty with efficiency as a measure is that it only takes into account increases in the number of cores, whereas increases in memory may be equally important. We argue in Section 7 that memory may become more of a bottleneck resource than cores. In [32], an alternative measure of scalability called the *iso-efficiency function* was introduced that measures the rate at which the problem size has to be increased with respect to the number of processors in order to maintain a fixed efficiency. This measure is also problematic, since size does not correlate well with difficulty in the case of ILPs, which makes choosing a test set even more challenging. Clearly, more effort is needed to develop coherent performance measures for this type of computation.

In the analysis below, we consider alternative measures of performance that provide a good indication of parallel performance in practice and do not require extensive testing with varying numbers of cores. These measure are based on the principle that the running time of a branch-and-bound algorithm is simply the product of the number of search tree nodes required to be processed (the size of the search tree overall) with the throughput rate, i.e., the number of search tree nodes processed per second per core. If both the size of the tree and the throughput remain constant as the number of cores is scaled up, then the result will be perfect efficiency. If efficiency drops, then it must be that either the size of the tree has increased (i.e., redundant work is being performed) or the throughput rate has dropped due to slowdowns resulting from any of the effects discussed in the previous section.

4.3 Sources of Variability

Unfortunately, even with the use of sensible measures of performance, rigorous experimentation on the scale we are proposing here is still extremely difficult due to the high variability experienced in execution of the solver, even when running on the same platform and solving the same problem. There are two main reasons for this variability: (1) lack of consistency in running times of single threads due to hardware effects and (2) lack of determinism in the order of execution of instructions with multiple threads. We examine each of these below.

Variability in Execution Time. In trying to improve the performance of a single CPU without increasing the clock speed, chip manufacturers have introduced some techniques that make it inherently difficult to measure performance consistently.

To begin with, all current multi-CPU systems employ *cache coherent non uniform memory access* (ccNUMA), which means that depending on where the specific memory to be accessed is located, the access time might vary by up to a factor of two. Because memory allocation may be different from one run to another, running times may vary due to these effects.

The latest generation of Intel CPUs employs a so-called *TurboBoost* functionality by which each CPU has an energy (or heat) budget. If this is not fully utilized, e.g., because some of the cores are idle, the clock speed of the working cores may be dynamically increased. This means that starting a second thread might decrease the clock speed of the first thread.

In addition to the physical computing cores, each CPU might have a number of logical cores through the use of a technique called *Hyper Threading* (HT) or *Simultaneous Multi Threading* (SMT). Typically, there are two (Intel Xeon), four (IBM Power7), or even eight (Sun T2) logical cores per physical core. Since these logical cores compete for the physical resources of the CPU, the total computing power depends very much on the load of the cores. For this investigation, both TurboBoost and hyper-threading were switched off when available.

Finally, certain operations related to I/O that one may perform in a sequential algorithm without penalty may not scale well due to specific design limitations of the architecture. Most notably, the cost of memory allocation and deallocation, which may be acceptable on a single core, does not scale linearly with the number of cores due to memory fragmentation effects and other related issues. This can become a very substantial scalability issue when a code performs frequent memory allocation and deallocation operations in processing a search tree node.

Non-deterministic Execution. For a single-threaded computer program, one would expect execution to be deterministic, i.e., two executions of a program in identical environments should be identical. However, this is not necessarily the case for a parallel program in a multi-threaded or distributed-memory environment. It is important to understand that when we say the behavior of the algorithm is non-deterministic, we do not mean that we expect the solver to get a different *result* (the result of the computation is the optimal solution *value*—the particular solution produced could in fact be different). What we mean by non-determinism is that the particular search tree explored in order to produce a given result may be different, resulting in a different set of computations, a different amount of work, and a different running time. When a program's execution is non-deterministic, performance measurement becomes difficult and must be done by sampling over multiple runs. For the foreseeable future, this will be too expensive on a million core system. Furthermore, debugging also becomes complex, as there is no guarantee that a bug will appear at the same stage of the program when run again.

It is possible to ensure deterministic behavior of a program by forcing all communication between the threads to happen at predefined synchronization points in the execution. However, this leads to increased idle time during the run and degrades performance. Depending on the size of the system and the variation in processing speed of the nodes, an extreme loss of efficiency is possible. While ensuring determinism for a shared memory system with a moderate number of threads can be done with acceptable efficiency [39], this is not the case for massive distributed memory systems. On a system as described above, it may already be extremely difficult to achieve an identical environment for even two runs. Assuming 8,000 PEs, there will be differences due to errors or genuine differences in hardware configuration of different PEs. As will be shown in the next section, the performance of current solvers can change dramatically depending on the number of threads used.

5 Solving on a Single PE (Shared Memory)

In solving an ILP on a single (shared-memory) PE, we have, in principle, two basic approaches available to us. We can either parallelize the processing of individual nodes by solving the LP relaxation in parallel using the barrier algorithm, or we can parallelize the search by processing multiple search tree nodes simultaneously. Because the simplex algorithm can be restarted so effectively, the latter approach has been adopted by all parallel ILP solvers currently available. Nevertheless, the former approach may have its place in certain situations, as we explore in Section 7.

A multi-threading execution mode is available in most commercial solvers. Implementation of this latter approach appears relatively easy, since the tree can be stored wholly in the memory and accessed by all threads (though contention becomes an issue as the number of threads increases). Putting aside the potential effects of ccNUMA memory access, one does not have to be concerned about moving data around to ensure it is available where it is needed, since the memory is shared. This means that the search strategy of the solver can remain relatively unchanged from the sequential case and, at least in principle, the impact of scaling on the size of the tree should be relatively minor.

5.1 Computational Experiments

As usual, the ultimate question is whether the solver performance actually does scale in practice. In [31], the average speed-up for all solvers when going from 1 to 12 threads was roughly a factor of 3. Since instances that can be solved within a few B&B-nodes generally will not scale, we examined five instances from the *Large Tree* subset of the MIPLIB 2010 [31], namely: `glass4, gmu-35-40, noswot, pigeon-10,` and `timetab1`. Using a 32 core Sun Galaxy 4600 equipped with eight Quad-Core AMD Opteron 8384 processors at 2.7 GHz and 512 GB RAM running Gurobi 4.5[2], solving these instances required between 97,223 nodes (noswot, 24 threads) and 665,106,682 nodes (gmu-35-40, eight threads). We can therefore assume there is ample opportunity to parallelize the processing of the B&B tree. Gurobi was used as a solver for this test because it was designed from scratch to utilize multi-core architectures. We assume therefore that its performance should be representative of the state of the art. All times given in the figures include the time for reading the instance, preprocessing, and solving the root node, though the total time for these tasks was in all cases much less than one percent of the total running time.

Figure 2 depicts the number of B&B nodes processed by the solver per thread per second. It is not surprising that `noswot` exhibits the largest decrease in this measure as the number of threads is increased, as solving `noswot` requires both the smallest number of B&B nodes and the smallest processing time per node. Accordingly, due to memory contention, the time required to update central data structures is likely slowing the solver down. The performance of `pigeon-10` is perhaps nearest to what one would expect, while the reason for the increase in performance for `gmu-35-40` and `glass4` when going from one to two threads is difficult to discern. The latter may be due to changes to the internal settings in the multi-threaded case, e.g., a higher tendency to dive or a reduced number of heuristic calls per thread. It could also be due to differences in memory allocation and better use of cache with multiple threads. In any case, we observe that with respect

[2] www.gurobi.com

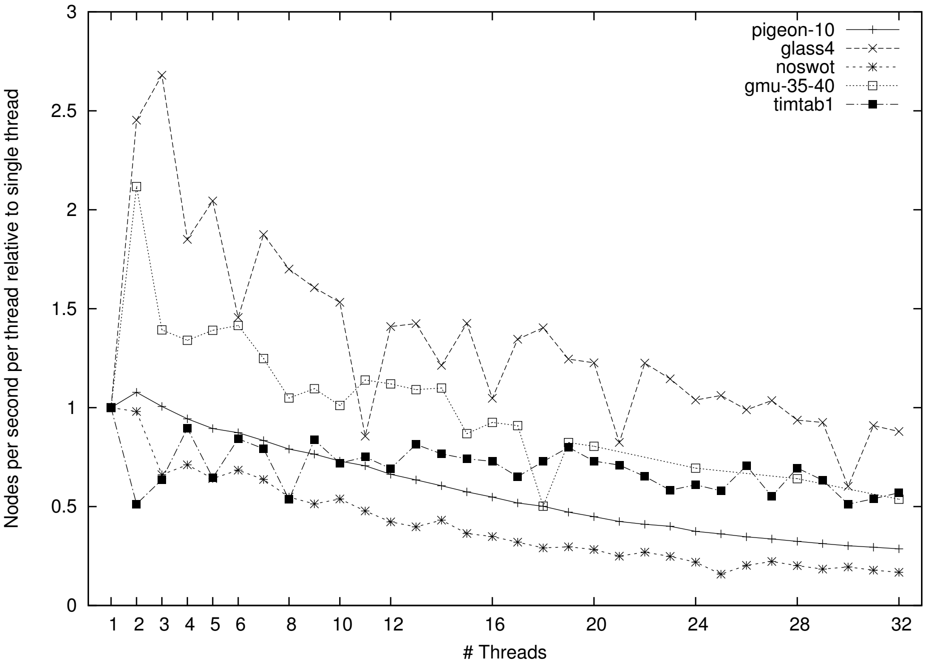

Figure 2: B&B nodes processed per thread per second

to this measure, the scalability properties of these instances look rather promising.

A natural question to ask at this point is how these results would scale to a 128-core system. If one simply replaces the existing 4-core processors by 16-core processors, the efficiency regarding the number of nodes per second per thread would approach zero, eventually coming to the point were adding threads reduces the total number of nodes computed per second. The reason is that for a given number of CPU sockets, the total memory bandwidth is limited. Once the memory interface is fully saturated, adding further cores to the computation will not have any beneficial effect. On the other hand, since this is a known bottleneck, each new system will usually have a higher memory bandwidth than its predecessor, as well as bigger cache memories[3] in order to reduce memory access requirements. Therefore, one could expect the picture to look pretty similar, but with appropriate scaling of the x-axis for the 128-core system.

Next, we investigate how this translates into parallel efficiency. A snapshot of typical real-world behavior is shown in Figure 3. Here, the behavior of `pigeon-10` is more or less what one would expect, while the achieved efficiency of the other instances looks more random and is usually poorer than hoped. When employing 32 threads, the best of these achieves an efficiency of 0.3, while the typical is more like 0.1. Worse than this, it is difficult to predict what the efficiency will be ahead of time. A partial explanation for this can be seen in Figure 4 (note the logarithmic scale of the y-axis). With the exception of `pigeon-10`, the number of nodes needed to solve an instance varies substantially with different numbers of threads and is often, but not always, higher than in the sequential case. Especially for `glass4` and `gmu-35-40`, the number of nodes needed can be as much as 30 times higher than in the sequential case. Even with no decrease in the number of nodes evaluated per second per thread, this increase in the total number of nodes evaluated is enough to substantially reduce efficiency. More information about performance variability of ILPs can be

[3] The upcoming Intel *Poulson* processor is going to have a 54 MB level 3 cache.

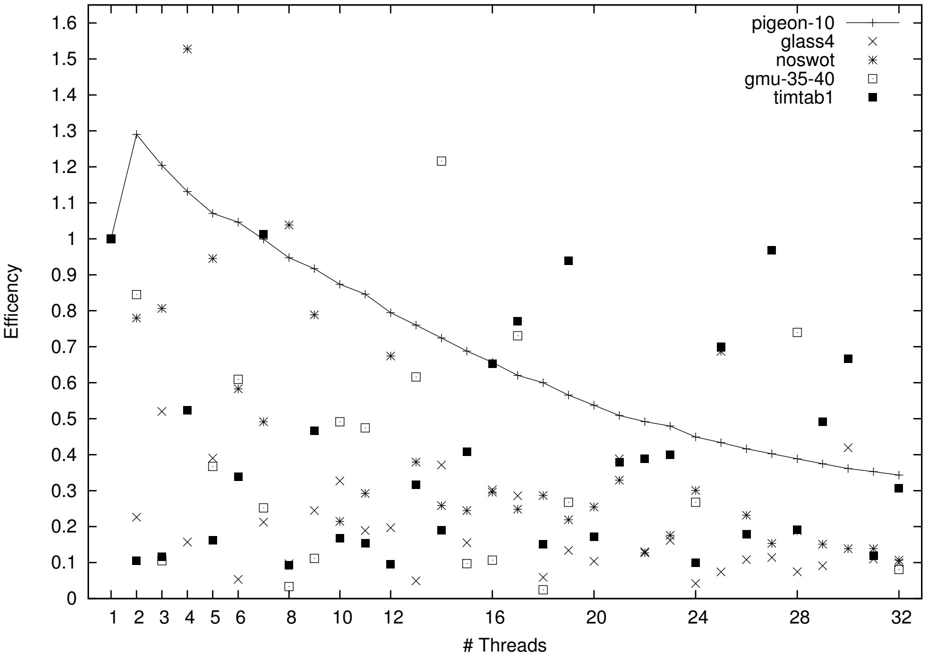

Figure 3: Solver efficiency by number of threads used

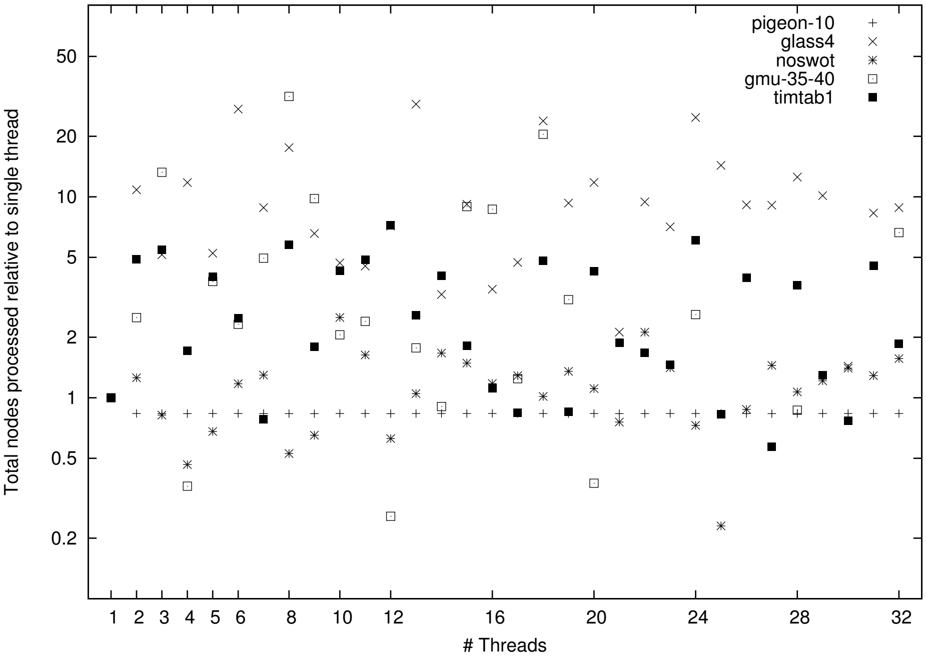

Figure 4: Total number of B&B nodes processed by number of threads

Table 3: Instances from MIPLIB2010 used as examples

Case		Name	Rows	Columns	Nonzeros	Status
(5)	Slow LPs	stp3d	159,488	204,880	662,128	solved
(4)	Large tree	reblock354	19,906	3,540	52,901	solved
(6)	Big	hawaii10-130	1,388,052	685,130	183,263,061	unsolved
(6)	Big	zib01	5887041	12,471,400	49,877,768	unsolved

found in [31].

5.2 Challenges

Although parallelizing branch and bound on a shared memory architecture seems straightforward, it is difficult to achieve efficiencies close to one. Among the reasons for this are effects such as contention for access to shared data structures and increased memory access times resulting from ccNUMA architectures. Scaling to higher numbers of threads seems possible for those instances for which many B&B nodes must be evaluated, provided we could ensure that the number of nodes needed is not generally much higher than in the sequential case. However, as the number of cores per PE scales up, we expect latencies to increase as the PE itself begins to look more and more like a distributed memory machine due to bottlenecks resulting for limited internal memory bandwidth.

6 Solution on Many PEs (Distributed Memory)

We now move to the question of whether we can effectively utilize several thousand PEs to solve one ILP. To get a feel for the answer to that question, we performed preliminary experiments with ParaSCIP [47] employing SCIP [2] as an ILP solver and using CPLEX 12[4] to solve the LP relaxations. ParaSCIP consists of a supervisor (load coordinator) system capable of maintaining the trunk of a B&B tree and distributing the solution of an ILP over a large number of PEs by use of the MPI communication protocol. Note that the variation in performance is higher in a distributed environment and singular results sometimes have rather complex explanations.

Four instances from MIPLIB 2010 will be used as examples in this and the next section. Table 3 shows some statistics describing these instances. The number in the Column labeled *Case* corresponds to the list of reasons for failing to solve an ILP given in Section 2.

6.1 Computational Experiments

Using ParaSCIP, the optimal solution to `reblock354` from MIPLIB 2010 was computed by exploring 41,918,266,856 B&B nodes. This took about 36 CPU years, delivered in less than one week of real time on 2,048 cores of the HLRN-II SGI Altix ICE 8200 Plus (ICE2) supercomputer, which consists of 960 Infiniband connected nodes with double quad-core Intel Xeon X5570 processors at 2.93 GHz with 48 GB memory each.

In the same way, it was possible to solve `stp3d`, introduced in MIPLIB 2003 [5], for the first time. Table 4 lists the solution times, comparing the use of 4,096 and 7,168 cores. `stp3d` clearly falls into category 5 of "hard-to-solve LPs" from the list given in Section 2, as can be seen from the rather

[4] www.cplex.com

Table 4: Solving times for stp3d on the HLRN-II ICE2 complex using distributed memory

Cores	4096	7168
Wall clock time [h]	42.32	30.68
Total CPU time [years]	19.79	25.08
Total Idle time [years]	0.80	1.82
Total Root time[years]	6.85	8.49
Nodes processed	9,779,864	10,328,113

low number of nodes. Using ParaSCIP, an efficiency of 0.79 was achieved when scaling from 4,096 to 7,168 cores, even though the parameter settings proved to be suboptimal.

It should be noted that for the stp3d run, the optimal solution was given as input. However, this appears to have much less influence on the solution time than one might think. In fact, there are astonishingly many cases in which having the optimal solution from the start actually leads to an increased solution time. The important point to note is that once the optimal solution is found, the order of the processing of the nodes does not matter (much) anymore.

Table 5 list those instances from the MIPLIB 2010 benchmark set which were solved by SCIP 2.0.1, but needed more than 10,000 B&B nodes. As can be seen in the table, in all cases considered, SCIP found the optimal primal solution before the dual bound reached the optimal value. For most instances, the optimal solution was found before even half of the nodes were enumerated.

6.2 Challenges

There are a number of difficulties with a distributed solution approach and we outline a few of these challenges here. The biggest of these is the substantial fraction of the running time occupied by the ramp-up and ramp-down phases (see discussion in [51]). There are approaches, such as, e.g., *racing ramp-up* [47], to utilize idle PEs during the ramp-up phase. As has been shown in [31], the performance of an ILP solver might vary randomly and substantially depending on the order of the constraints and variables in the problem and also on the number of threads employed. This effect is of course even stronger if one is changing the basic parameters of the solver, such as frequency of heuristics or number of cutting planes generated. The idea of *racing ramp-up* is to explore multiple search trees with different solver settings in parallel until some stopping criterion, such as a threshold for generation of a sufficient number of nodes, is reached. It is then decided which of the trees generated so far should be retained for the next phase of computation. The nodes of this tree are distributed among the PEs, while all the other results, with the exception of primal solutions, are discarded. So far, however, this and other approaches have not proven to be effective enough to make up for the reduction in subproblems solved per thread per second in the initial parts of the algorithm.

Connected to this is the question of how to select the next node to process in the tree. In the beginning, it might be helpful to use a selection that leads to a wide tree, while switching later in the processing to a depth-first search (DFS) to save memory, as necessary. The way the subproblems are split has a huge impact on the shape of the generated tree. While in the sequential case, it can be useful to have an “uneven” split that leads to a quick fathoming of one of the branches, this is unlikely to be helpful during the ramp-up phase. The impact of the branching-rule on the overall performance of an B&B-algorithm, in both the sequential and parallel cases, cannot be underestimated. This has been an area of intensive research (see, e.g., [4, 3, 34]).

Table 5: % number of nodes processed until first solution and optimal solution

Instance	Total nodes	% nodes at first	% nodes at optimum
mik	415149	0.0	0.0
iis-100-0-cov	103865	0.0	0.2
noswot	605006	0.0	0.4
n4-3	81505	0.0	1.6
neos-1109824	10792	0.1	1.9
qiu	14452	0.0	2.6
aflow40b	278717	0.0	4.6
pg5_34	257034	0.0	5.8
neos-916792	67445	0.1	14.7
dfn-gwin-UUM	14491	0.0	15.8
eil33	11129	0.0	18.7
ran16x16	344269	0.0	23.3
roll3000	593902	0.0	27.7
reblock67	139635	0.0	28.1
enlight13	622598	30.1	30.1
bienst2	89641	0.0	30.3
binkar10_1	199430	0.0	39.2
rococoC10-001000	462506	0.0	44.5
iis-pima-cov	11113	0.0	45.9
mcsched	23111	0.0	53.5
neos-1396125	49807	9.1	55.3
mine-90-10	56177	0.0	56.5
timtab1	699973	0.0	60.5
unitcal_7	12264	0.0	63.9
harp2	319153	0.0	79.9
rocII-4-11	27610	0.4	85.5
ns1830653	47032	3.1	85.8

Ramp-down is usually less critical and, as opposed to ramp-up, profits from algorithmic developments that make the tree smaller. Nevertheless, both situations typically decrease efficiency as the number of threads increases.

Another difficulty is that PEs can run out of memory. Using many cores, a single PE might produce an excessive number of open B&B nodes when no subtree can be fathomed. We experienced this, for example, in trying to solve `dano3mip`. Writing node files is not feasible, as this would require writing several petabytes to disk. For the same reason, a transfer of the nodes back to the load coordinator is also not realistic. A possible solution is to switch to iterated DFS as the node selection strategy to limit the number of newly created nodes. While this will increase the number of nodes processed per second, it generally leads to a higher number of total nodes (see [2] for some details). The bottom line is that the total number of open nodes is limited (even with the increase due to parallelism), and this can be problematic.

The third problem comes from the fact that for the foreseeable future, machines with one million cores will be expensive and somewhat unreliable. This means that computing time will be limited and runs may have to be interrupted. After having run two days on a system with a million cores, one would not be willing to throw the results of the computation away. To remedy this, some way of checkpointing is necessary. However, as we pointed out previously, writing descriptions of all open nodes to disk is likely to be excessive. One solution is to write only the roots of subtrees, as stored by the load coordinator. In this case, some work is lost, but not all. The effectiveness of this depends very much on the instance.

The biggest challenge is to decide which instances fall into the category of ILPs for which this type of computing is appropriate. So far, it has proven difficult to estimate the number of B&B nodes that will be needed to solve an instance early in the solution process [19, 40]. It remains very unclear how many more instances could be solved if 10, 100, or 1000 times the number of nodes could be evaluated. The instances that could likely profit the most from additional node evaluations are those instances for which the LPs are hard to solve and the number of nodes processed per second is the limiting factor. Here, using a million cores clearly offers a viable approach.

7 Solution of Very Large ILPs

We have assumed thus far that the size of an instance is small enough to fit into the memory of a single PE, which limits us a priori to instances of a certain maximum size. On the other hand, we are able to process a vastly larger number of B&B nodes per second in parallel, therefore enabling us to solve instances requiring many more total nodes. In addition, we are able to manage a much larger number of open nodes than on a single PE.

An obvious question that now arises is what to do with instances that are too big to fit into the memory of one PE. If we assume that one PE is big enough to solve a single LP relaxation, the revised dual simplex algorithm is usually the method of choice for solution of LP relaxations. However, there are cases for which the barrier algorithm or even some other alternative are more appropriate choices. One such case is when the instance is very large, perhaps even too large to solve on a single PE. In this section, we address this possibility.

Table 6: Estimated comparison of simplex and barrier algorithms for solving LPs

	Simplex	Barrier
Basic speed	1	0.6
Warm-start speed-up	5–500×	1 (2-10) ×
Parallel speed-up	1 (2) ×	16+ ×
Needs crossover	no	yes
Memory requirement	1×	up to 10×

Table 7: Performance of simplex and barrier algorithms on large instances

Instance	Solver	Method	Threads	Mem [GB]	Iterations	Time [s]
hawaii10-130	CPLEX	Simplex	1	6	55,877	762
	Gurobi	Simplex	1	21	175,782	6,967
	CPLEX	Barrier	32	39	130	47,933
	Gurobi	Barrier	32	56	191	43,037
zib01	CPLEX	Simplex	1	7	>15,152,000	>435,145
	Gurobi	Simplex	1	10	10,833,995	166,177
	CPLEX	Barrier	32	38	28	4,953
	Gurobi	Barrier	32	51	34	6,455

7.1 Simplex Versus Barrier

The main advantages of the simplex algorithm are its very good numerical stability, its excellent warm-start capability, and the fact that it computes vertex solutions. The biggest drawback is that the simplex algorithm seems to be inherently sequential. While there has been some success in parallelization in special cases [11, 30], such as when the number of columns is much bigger than the number of rows, all attempts at general-purpose parallelization have so far failed [50, 26]. This is in contrast to the barrier algorithm, which parallelizes quite well. The barrier algorithm, however, lacks warm-start capabilities. A speed-up factor of two from warm-start seems to be consistently possible, but ten is the maximum speed-up that seems remotely possible at the moment (see, e.g., [23, 53, 8, 29]). Even worse, the solution provided by the barrier algorithm is not a vertex solution and since such a solution is needed for generation of several of the most common classes of valid inequalities, a crossover [38, 14] procedure has to be performed. This crossover is itself a simplex-type algorithm and therefore sequential. In [45], the authors estimate that the crossover takes up to 20% of the total solution time when using 12 threads. Finally, the memory consumption of the barrier algorithm is higher than that of the simplex algorithm. In certain cases, it may need as much as ten times the amount of memory. Table 6 shows a summary comparison of the simplex and the barrier algorithms based on the above discussion. Numbers in parentheses indicate what is considered possible without having been generally implemented so far.

The question of whether the simplex algorithm or the barrier algorithm is the faster algorithm to solve LPs is not new and is difficult to answer. For small instances, the simplex algorithm is often faster, while for larger instances, the barrier algorithm seems to hold an edge. There are two important points. First, it seems the two algorithms are complementary in the sense that the barrier algorithm typically performs quite well on instances for which the simplex algorithm has difficulties and vice versa. Second, there is still no reliable way to tell in advance which of the two will be faster. To illustrate, we tested both algorithms on the instances `hawaii10-130` and `zib01` with CPLEX 12.2 and Gurobi 4.5. The results are shown in Table 7. The instances were

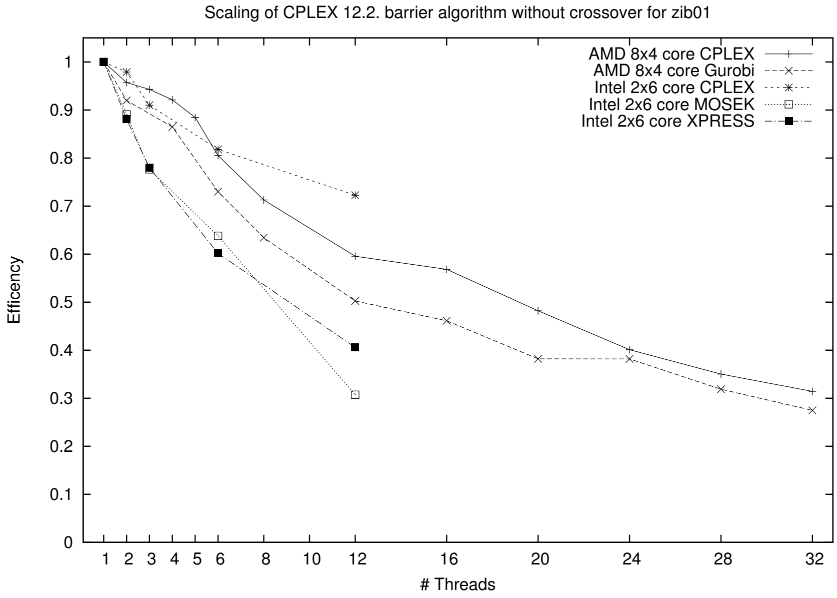

Figure 5: Efficiency of solving `zib01` with different barrier solvers / including ordering / no crossover

deliberately chosen to show opposing aspects of algorithm performance. We compare the single-thread performance of the simplex algorithm to the 32-thread performance of the barrier algorithm. As can be seen, on `hawaii10-130`, the simplex algorithm is the clear winner, while on `zib01`, the barrier algorithm is much faster.

A pivotal question, however, is how much it helped the barrier algorithm to be able to use 32 threads. We solved instance `zib01` using several solvers[5] with different numbers of threads. The runs with up to 32 cores were performed on the same SUN Galaxy 4600 as mentioned before. For the runs with 12 cores, a system with two hexa-core Intel Xeon X5680 CPUs at 3.33 GHz with 32 GB RAM was used. Figure 5 depicts the results. As can be seen, the barrier algorithm scales well in all cases with up to four threads, while with CPLEX, 16 threads are required to reduce the efficiency to 0.5. Nevertheless, with 32 threads, we are down to an efficiency of about 0.3. Interestingly, this is somewhat similar to what we saw in Figure 3 for solving ILPs. One of the reasons could be the limited memory bandwidth of the machine.

Looking at the results of several solvers for the MIPLIB 2010 benchmark set, one can observe that the geometric mean of the number of simplex iterations needed to solve the root LP is about 1,500 and the number of iterations needed for reoptimizing the LP relaxation is about 60. Thus, we have an average speed-up factor of roughly 25, which is less than what is commonly thought. One must, however, keep in mind that a low speed-up factor is not necessarily an indication of poor performance of the warm start, as it may also be due to a low number of simplex iterations in the initial solve. In those cases, the speed-up achieved for reoptimization might be lower than average, although the simplex algorithm is performing very well. For `gmu-35-40, noswot, pigeon-10`, and `timetab1`, the number of simplex iterations to reoptimize is typically below ten. For instances that can be solved with such a small number of iterations, the simplex algorithm is usually already faster than the barrier algorithm for the initial solve. Empirically, this behavior changes for larger

[5] CPLEX 12.2.0.2 (www.cplex.com), Gurobi 4.5.0 (www.gurobi.com), MOSEK 6.0.0.106 (www.mosek.com), XPRESS 22.01.03 (www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx)

instances, however. The barrier algorithm becomes faster at solving the root node in comparison, but the speed-up ratio for the simplex algorithm increases.

The above leaves two basic options: either use the barrier algorithm in order to profit from the parallelization (but give up the benefit offered by reoptimization) or use the simplex algorithm and leave 127 of our assumed 128 cores idle on each PE, assuming that we can only handle one LP relaxation on a single PE due to memory limitations. If we are in fact limited to solving a single LP relaxation on each PE at any one time, the simplex algorithm might be the only choice, given its lower memory requirements. However, this gives little hope of utilizing the additional cores efficiently. Though one typically endeavors to utilize all available CPU cycles and leave some memory idle, an abundance of cores may require getting used to the idea of leaving the *cores* idle and utilizing the *memory* instead. In other words, it may turn out that the memory, not the available cycles, may become the bottleneck. On the other hand, as we show in Section 8, one should keep in mind that for any instance so large that only one core can be utilized, the time needed to solve a single LP relaxation might be prohibitive to the overall solution process anyway.

Given that only one LP can be solved on a PE, the additional memory to run an ILP solver with iterated DFS node selection is small. Therefore, we can assume this works as described in the previous section, though this might lead to an increased number of B&B nodes.

7.2 Alternative Algorithms for LP

In addition to the simplex and the barrier methods, there are a number of algorithms [10], e.g., Lagrangian relaxation [21], the volume algorithm [7], and bundle methods [27], that can compute lower bounds on the LP optimum and might even converge to it. They usually give lower bounds very quickly, do not need much memory, and are quite suited for large-scale instances. In fact, there have been a number of successes for special applications, e.g., [15]. Nevertheless, in the general case, it seems difficult to use these as a replacement for the simplex or the barrier algorithms. They typically produce no primal feasible solution and the dual solution is not a vertex solution in general. There might be some hope of using these alternatives for binary problems, but they seem unrealistic for solving general ILPs without being able to exploit special structure.

8 Conclusions

The only rules that really matter are these:
what a man can do and what a man can't do.
Captain Jack Sparrow

8.1 Summary

We conclude by summarizing these three basic cases:

The instance is small enough to run multi-threaded on a PE. Based on the results of Section 5, an efficiency of $E_c = 0.2$ for a 128-core ILP solver can reasonably be assumed. This means we can achieve a speed-up of roughly 25. From Section 6, we assume an efficiency for distributing over many PEs of $E_{pe} = 0.1$, which means we achieve roughly a speed-up factor of 800 on 8,000

PEs in steady state. Together, we have a speed-up of

$$E_c \times N_c \times E_{pe} \times N_{pe} \approx 20,000$$

compared to a sequential run. Since we can run easily on one PE, this means, roughly, we can evaluate as many nodes on the million core machine in one day as we would be able to compute in two years on one PE in steady state. For instances with very large trees, the ramp-up and ramp-down phases of the computation should have an acceptable impact on efficiency.

The LP relaxation of the instance can be solved on a single PE. In this case, we face the question of whether to use the simplex or the barrier method. As described in Section 7, this depends very much on the instance and the available memory. We can assume the barrier algorithm with crossover to run at an efficiency of 0.2, giving us a speed-up of 25 on a 128-core machine. This is likely about equivalent to the speed-up we get from the simplex algorithm running sequentially, due to the warm start. We end up with a speed-up of

$$E_{pe} \times N_{pe} \approx 800.$$

The difference with respect to running on one PE is the same as before, but we compute about 25 times fewer nodes per second. Furthermore, because solving an LP that needs half a terabyte of RAM will likely take at least ten minutes, the whole machine will only be able to compute at most 15 B&B nodes per second. Hence, the ramp-up and ramp-down phases are likely to impact efficiency further.

Solution of the LP relaxation has to be distributed. In this case, a distributed barrier algorithm would currently be the only choice. There are some implementations for distributed memory linear, non-linear and semi-definite programming solvers, e.g. [25, 16, 52, 28, 36, 20]. but to our knowledge, none of these is suitable for obtaining an estimate of the efficiency of an optimized distributed solver for pure linear programs. Therefore, we assume an efficiency of 0.1 for this method. We then end up with

$$N_{pe}/N_{lp} \times 0.1 \times E_{pe} \approx 10,$$

assuming we need eight PEs per LP. Even if we assume that we could solve one distributed LP in 15 minutes, we would have an equivalent performance of 1 B&B node per minute. This would allow us to compute a few thousand nodes before we run out of computing time. Also, there would be very severe ramp-up problems.

Furthermore, without a distributed simplex algorithm we will have no cross-over procedure, which will hamper the generation of cutting planes and will lead to an increased number of B&B nodes necessary to solve the problem. In the same sense, many of the primal heuristics would either have to be implemented in a distributed fashion, with the drawback that they rely on solving special LP subproblems. Many diving heuristics are pretty much out of the question with this approach.

In summary, since the solution methodologies for ILPs rely very heavily on the ability to solve LPs quickly and it therefore seems highly questionable whether a general-purpose ILP solver for instances that need distributed solution of LPs is useful. In those cases, it seems much more promising to implement specialized methods.

Table 8: Inabilities and their cures

Symptom	HW cure	SW cure
Slow LPs	Faster cores	LP algorithm improvement
Many nodes	More cores	ILP algorithm improvement
Big instance	More memory	Different LP algorithm

8.2 Open Questions

Table 8 gives an overview of what change would help to solve which challenges. Faster cores would help in all cases (this was depended upon for decades). Increased numbers of cores may help in cases where the LP solution times are slow and may also help with very large instances, but the latter case is hampered by memory bandwidth constraints, the requirement for sequential LP solution algorithms, and ramp-up/ramp-down issues. Better LP algorithms would help a lot for big instances.

So what should we take away from all of this? Unfortunately, the effect of the development of algorithms for ILPs that are considered "better" in the traditional sense of sequential computing time is usually a reduction in the size of the tree, which actually results in less efficient parallelization. The path to development of more efficiently parallelizable algorithms is thus very unclear. One thing is clear, however. Straightforward parallelization of algorithms developed originally for sequential execution will have limited scalability. To move forward, we must begin to think out of the box. It may be that the key is to embrace a completely new solution paradigm that breaks from the traditional strategies we have used quite successfully until now. Rather than drawing any solid conclusions, we end this investigation by posing some challenge questions that may help move us in new directions.

- Most current ILP solvers are designed under the assumption that computing cycles are the main bottleneck. This is not generally true any more if a million cores are available. What implications does this have on the solver algorithms?

- During the solution of an ILP, there are typically several phases, e.g., the time until the first feasible solution is found or the time until an optimal solution has been discovered but not yet proved optimal. Especially in the case of large enumeration trees (item 4 from the list given in Section 2), the distribution of time between the phases might substantially change. Again, the question arises, what are the implications of this for the ILP solver?

- Typically, the time until 1 million active B&B nodes are available is considerable. A similar effect occurs at the end of the computation. What should we do during these so-called *ramp-up* and *ramp-down* phases to utilize the available computing resources?

- Decomposition has always been a topic with much potential that is difficult to realize. Since very large ILP instances often have structure exploitable by decomposition, decomposition may still be a promising direction to explore.

- The main obstacle to solving bigger ILPs is the solution of the LP relaxations. The simplex method does not scale with the number of cores and the barrier algorithm is not well suited for re-solving LPs as they occur in B&B based ILP solvers. Improvements in this area will directly translate to an increased ability to solve bigger instances.

9 Acknowledgements

The work of Thorsten Koch and Yuji Shinano was supported by a Google Research Grant. We would like to thank the HLRN-II for providing computation time and the HPC department at ZIB for their efforts to make runs with 7,168 cores possible. Finally, we would like to thank Hans Mittelmann for providing the computations on the 12-core system. We would also like to thank the anonymous referees for their insightful remarks.

References

[1] K. Aardal, R. Weismantel, and L. A. Wolsey. Non-standard approaches to integer programming. *Discrete Applied Mathematics*, 123:5–74, 2002.

[2] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] T. Achterberg and T. Berthold. Hybrid branching. In W.-J. van Hoeve and J. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer, 2009.

[4] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[5] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):361–372, 2006.

[6] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2007.

[7] F. Barahona and R. Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87:385–399, 2000.

[8] H. Benson and D. Shanno. An exact primal-dual penalty method approach towarmstarting interior-point methods for linear programming. *Computational Optimization and Applications*, 38:371–399, 2007.

[9] T. Berthold and M. E. Pfetsch. Detecting orbitopal symmetries. In B. Fleischmann, K. H. Borgwardt, R. Klein, and A. Tuma, editors, *Operations Research Proceedings 2008*, pages 433–438. Springer, 2009.

[10] D. Bienstock. *Approximation Algorithms for Linear Programming: Theory and Practice.* CORE Lecture Series. Core, UCL, Belgium, 2001.

[11] R. Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12:45–56, 2000.

[12] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[13] R. E. Bixby. Lectures about LP and MIP solving at Combinatorial Optimization at Work II, 2009.

[14] R. E. Bixby and M. J. Saltzman. Recovering an optimal basis from an interior point solution. *Operation Research Letters*, 15:169–178, 1994.

[15] R. Borndörfer, A. Löbel, and S. Weider. A bundle method for integrated multi-depot vehicle and duty scheduling in public transit. In M. Hickman, P. Mirchandani, and S. Voß, editors, *Computer-aided Systems in Public Transport*, volume 600 of *Lecture Notes in Economics and Mathematical Systems*, pages 3 – 24, 2008.

[16] T. F. Coleman, J. Czyzyk, C. Sun, M. Wagner, and S. J. Wright. ppcx: Parallel software for linear programming. In *Proceedings of the Eighth SIAM Conference on Parallel Processing in Scientific Computing.* SIAM, 1997. http://www.cs.cornell.edu/Info/People/mwagner/pPCx/paper.ps.

[17] W. Cook, T. Koch, D. Steffy, and K. Wolter. An exact rational mixed integer programming solver. In *Proceedings of the* 15^{th} *Conference on Integer Programming and Combinatorial Optimization*, pages 104–116. Springer, 2011.

[18] W. Cook, T. Rutherford, H. E. Scarf, and D. Shallcross. An implementation of the generalized basis reduction algorithm for integer programming. *ORSA Journal on Computing*, 5(2):206–212, 1993.

[19] G. Cornùejols, M. Karamanov, and Y. Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.

[20] F. E. Curtis, O. Schenk, and A. Wächter. An interior-point algorithm for large-scale nonlinear optimization with inexact step computations. *SIAM Journal on Scientific Computing*, 32(6):3447–3475, 2010.

[21] M. L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 50(12):1861–1871, 2004.

[22] G. Gamrath and M. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2010.

[23] J. Gondzio. Warm start of the primal-dual method applied in the cutting-plane scheme. *Mathematical Programming*, 83:125–143, 1998.

[24] M. Grötschel, M. Jünger, and G. Reinelt. A Cutting Plane Algorithm for the Linear Ordering Problem. *Operations Research*, 32(6):1195–1220, 1984.

[25] A. Gupta and V. Kumar. A scalable parallel algorithm for sparse cholesky factorization. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 793–802, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[26] J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7:139–170, 2010.

[27] C. Helmberg and K. Kiwiel. A spectral bundle method with bounds. *Mathematical Programming*, 93:173–194, 2002.

[28] I. D. Ivanov and E. de Klerk. Parallel implementation of a semidefinite programming solver based on CSDP in a distributed memory cluster. Technical Report CentER Discussion Paper 2007-20, Tilburg University, The Netherlands, 2007.

[29] E. John and E. A. Yildirim. Implementation of warm-start strategies in interior-point methods for linear programming in fixed dimension. *Computational Optimization and Applications*, 41:151–183, 2008.

[30] D. Klabjan, E. L. Johnson, and G. L. Nemhauser. A parallel primal-dual simplex algorithm. *Operations Research Letters*, 27(2):47–55, 2000.

[31] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3:103–163, 2011.

[32] V. Kumar and V. N. Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16:501–519, 1987.

[33] D. Levinthal. Performance analysis guide for Intel core i7 processor and Intel Xeon 5500 processors.

[34] A. Mahajan and T. K. Ralphs. Experiments with branching using general disjunctions. In *Proceedings of the Eleventh INFORMS Computing Society Meeting*, pages 101–118, 2009.

[35] A. Mahajan and T. K. Ralphs. On the Complexity of Selecting Disjunctions in Integer Programming. *SIAM Journal on Optimization*, 20(5):2181–2198, 2010.

[36] Y. Makoto and K. Fujisawa. Efficient parallel software for large-scale semidefinite programs. In *Proceedings of the 2010 IEEE Multi-conference on Systems and Control*, Sept. 2010.

[37] F. Margot. Symmetry in integer linear programming. In M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, editors, *Fifty Years of Integer Programming: 1958–2008*, pages 647–686. Springer, 2010.

[38] N. Megiddo. On finding primal- and dual-optimal bases. *ORSA Journal on Computing*, 3(1):63–65, 1991.

[39] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.

[40] O. Y. Özaltin, B. Hunsaker, and A. J. Schaefer. Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. *INFORMS Journal on Computing*, 23(3), 2011.

[41] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, February 1991.

[42] W. Paper. SGI Altix global shared memory performance and productivity breakthroughs for the SGI Altix UV, Oct. 2010. http://www.sgi.com/pdfs/4250.pdf.

[43] C. Phillips, J. Eckstein, and W. Hart. Massively parallel mixed-integer programming: Algorithms and applications. In M. Heroux, P. Raghavan, and H. Simon, editors, *Parallel Processing for Scientific Computing*, pages 323–340. SIAM Books, Philadelphia, PA, USA, 2006.

[44] E. Pinheiro, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In *Proceedings of the* 5^{th} *USENIX Conference on File and Storage Technologies (FAST'07)*, Berkeley, CA, 2007. USENIX Association.

[45] E. Rothberg. Barrier is from mars, simplex is from venus, 2010. Talk given at *What a pivot–workshop honouring the 65th birthday of Bob Bixby* in Erlangen, Germany.

[46] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 193–204. ACM, 2009.

[47] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP – a parallel extension of SCIP. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, February 2012.

[48] L. A. Wolsey. *Integer programming*. Wiley-Interscience, 1998.

[49] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995.

[50] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

[51] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Computational experience with a software framework for parallel integer programming. *INFORMS Journal on Computing*, 21:383–397, 2009.

[52] M. Yamashita, K. Fujisawa, and M. Kojima. Sdpara : Semidefinite programming algorithm parallel version. *Parallel Computing*, 29:1053–1067, 2003.

[53] A. Yildirim, Stephen, and S. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization*, 12:782–810, 2000.