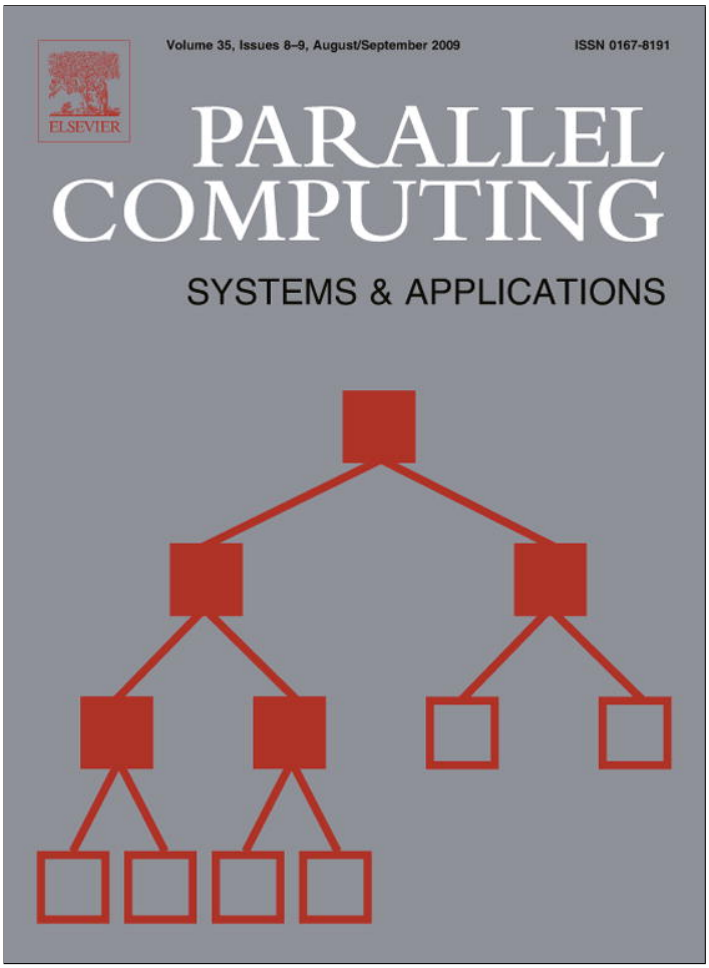

Parallel Computing 35 (2009) 455–474

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Performance implications of synchronization structure in parallel programming

Arturo González-Escribano [a], Arjan J.C. van Gemund [b,*], Valentín Cardeñoso-Payo [a]

[a] Dept. de Informática, Universidad de Valladolid, E.T.I.T. Campus Miguel Delibes, 47011 Valladolid, Spain
[b] Faculty of Electrical Engineering, Mathematics, and Computer Science, P.O. Box 5031, NL-2600 GA Delft, The Netherlands

ARTICLE INFO

Article history:
Received 14 September 2007
Received in revised form 11 June 2009
Accepted 24 July 2009
Available online 14 August 2009

Keywords:
Parallel programming models
Task graphs
Performance prediction

ABSTRACT

The restricted synchronization structure of so-called structured parallel programming paradigms has an advantageous effect on programmer productivity, cost modeling, and scheduling complexity. However, imposing these restrictions can lead to a loss of parallelism, compared to using a programming approach that does not impose synchronization structure. In this paper we study the potential loss of parallelism when expressing parallel computations into a programming model which limits the computation graph (DAG) to series–parallel topology, which characterizes all well-known structured programming models. We present an analytical model that approximately captures this loss of parallelism in terms of simple parameters that are related to DAG topology and workload distribution. We validate the model using a wide range of synthetic and real-world parallel computations running on shared and distributed-memory machines. Although the loss of parallelism is theoretically unbounded, our measurements show that for all above applications the performance loss due to choosing a series–parallel structured model is invariably limited up to 10%. In all cases, the loss of parallelism is predictable provided the topology and workload variability of the DAG are known.

1. Introduction

The large complexity of contemporary computer architecture poses great challenges to designers of parallel applications as well as the compilers that must shield programmers from the many specific details of the architecture. One of the main problems in high-performance computing is the lack of performance predictability, as the complex interplay between parallel program and parallel machine can no longer be overseen by user and/or compiler, often leading to disappointing performance surprises.

Two major factors underlying the lack of performance predictability are memory hierarchy and parallelism. Although memory hierarchy is a significant factor, it is not particular to parallel programming where parallelization decisions can also greatly influence performance. Despite advances in data-parallel programming and associated compiler technology [1,33,31,19], explicit parallel programming models, such as message-passing (e.g. [13]), or direct thread-programming (e.g. [10]) are still the only programming models guaranteed to deliver the hardware's raw parallelism. Unfortunately, programming in terms of these totally unrestricted coordination models is error-prone and inefficient, as the performance of parallel programs involving complex, spaghetti-type synchronizations is extremely difficult to analyze by humans and compilers [5,20,37]. As a result, a task or data partitioning and machine mapping that seemed to make sense can induce an

* Corresponding author.
E-mail address: a.j.c.vangemund@tudelft.nl (A.J.C. van Gemund).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.07.002

unforeseen performance penalty, with deadlock often just behind the corner. The central question, addressed in this paper, is whether these, sometimes rather intimidating, programming models are really necessary to attain optimum performance.

The choice of parallel programming model is an important factor in a wide-spread acceptance of high-performance computing. Acceptance of a programming model greatly depends on conflicting requirements. On the one hand, the model must conveniently abstract away from machine architecture complexity and offer simplicity. On the other hand, the model must provide sufficient synchronization flexibility to enable accurate expression of the data dependencies of a particular algorithm, and to derive efficient implementations which exploit the native unstructured hardware and lower-level programming tools.

Some examples of this trade-off can be found, e.g., in OpenMP [11] or UPC [17], amongst others, where the programmer still faces many decisions about data layout and sometimes unstructured synchronization. Other proposals impose a *restricted* synchronization model [51] (either provided as stand-alone programming model, or *embedded* within a less restricted programming model). The predominant coordination model found in these restricted programming models is *nested parallelism* (aka Divide & Conquer). Examples include BSP [55] and nested-BSP (e.g., NestStep [36], PUB [9]), approaches based on Skeletons (e.g., SCL [14], Frame [12]), Cilk [7], Satin [44], BMF [50], ASSIST [3], SPC [22], and SPC-XML [29]. In these programming models the parallel constructs have the same restricted semantics as the nested forms of the well-known `cobegin–coend` primitives [4]. In terms of the associated DAG (directed acyclic graph, the computation graph, whose nodes represent tasks and directed edges precedence relations) nested-parallelism restrict the DAGs to the important class of *series–parallel* (SP) DAGs [54,41]. They come with a firm theoretical foundation [40] and reduced complexity for many combinatorial problems [52].

Compared to unstructured (non-SP, NSP) models, the attractive properties of these so-called SP programming models, include improved analyzability [40,39], cost estimation efficiency [22,23,47,50], mapping/scheduling efficiency [8,18], and most notably, parallel programming efficiency [51]. While the first two properties are essential to (semi-) automatic compilation and mapping approaches aimed to increase portability and performance, the last property is often regarded as a particular selling point of structured parallel programming models. To illustrate the difference in programming complexity consider a simple macro-pipeline computation [42] associated with the pipelined processing of a stream of N data d_i where d_i might be a media sample that is to be processed in P consecutive stages (e.g., a codec). The sequential algorithm can be concentually expressed by the following pseudo code:

```
for i = 1...N do
  for j = 1...P do
    d[i, j + 1] = f_j(d[i, j])
```

where f_j denotes the function applied at the j'th stage, and d_{ij} denotes d_i at processing stage j. Note that obvious performance optimizations, such as the use of, e.g., temporary (register) storage instead of a statically allocated array `d[*,*]`, have been omitted for simplicity.

As the execution time for each function f_j may differ it is worth-while to consider an NSP programming approach that maximizes potential parallelism, such as the following parallel pseudo code

```
forall j = 1...P do
  for i = 1...N do
    wait(lock[j]);
    d[i, j + 1] = f_j(d[i, j])
    signal(lock[j + 1]);
```

The `forall` construct has the usual fork/join (cobegin/coend) semantics. In this algorithm the P functions f_j execute in parallel, only synchronized by the data dependencies, implemented by the `lock` semaphores (or message-passing constructs if a message-passing model were used). This loosely synchronized ensemble allows each function f_j to operate as soon as input data comes available, exploiting maximum parallelism. Note that many unnecessary details, such as pertaining to the boundary conditions, have been omitted for simplicity.

In turn, we consider an SP programming solution, yielding the following parallel pseudo code

```
for i = 1...N do
  forall j = 1...P do
    d[i, j + 1] = f_j(d[i, j])
```

where the entire pipeline f_j executes synchronously (lockstep). In contrast to the earlier, NSP version, this SP version only exhibits simple fork-join parallelism, leading to simple and intuitive code (note that for more complex algorithms the difference in programming complexity quickly increases due to the additional synchronization spaghetti for the NSP version). While simple, its performance may suffer compared to the NSP version when the execution times of f_j differ, i.e., when the pipeline is not load balanced (the performance loss depends on N and P).

1.1. Performance implications

While the above programming example illustrates that parallel programming efficiency may be improved, the restrictions imposed by the limited synchronization model offered by the so-called "structured" parallel programming models may adversely affect performance. This is due to the fact that the additional synchronizations that may come with imposing SP synchronization structure may result in an increase of the length of the *critical path* of the parallel program, thus limiting the amount of parallelism that can be exploited. For example, consider two example computations, *viz.* (1) the regular, macro-pipeline computation which was used earlier as an example, and (2) an irregular computation originating from a Finite Element Method (FEM) problem [38], whose associated task (dependency) graphs are given by the NSP DAGs shown on the left-hand sides of Figs. 1 and 2, respectively. The right-hand side of the figures depict SP DAGs, that are associated with an alternative, SP programming solution, where the dashed lines represent the additional barrier-synchronizations due to the SP programming model. Note that the two DAGs in Fig. 1 correspond to the NSP and SP algorithms given earlier.

As can be seen from the figures, for unchanged workloads the critical path length of the SP version is greater or equal to the NSP version. For an evenly balanced task workload distribution the SP version still has a critical path length equal to the original NSP version. In practice, however, some workload imbalance is not uncommon. As mentioned earlier, in media processing pipelines the individual task workloads are often content-dependent, while in FEM problems the task workloads depend on the structural decomposition technique, and can vary considerably. As a consequence, the performance loss may be non-negligible. Consider the following pathological workload distribution where the black nodes represent a computation workload of, e.g., 1 time unit, while the white ones represent *zero* workload. While in the NSP cases the critical path length equals 1 time unit, in the SP versions the critical path length increases to 4, and 3, respectively (despite the fact that the SP versions even represent the best – most parallel – SP versions possible for this pathological workload distribution). Despite the negligible probability of such workload distribution, there clearly exists a class of DAG topologies and workload distributions for which the performance loss of choosing an SP programming model may be appreciable (a factor of 3 and 4 in the above examples), as also illustrated by the additional examples that appear throughout the paper. To the best of our knowledge there has been no systematic study on which parallel computations carry this potential problem, and to what extent a performance loss is incurred when mapping to a real platform.

To date, the choice whether to use a structured, or an unstructured, yet more versatile parallel programming model is often determined by personal experience and folk wisdom, rather than by quantitative, scientific study. In this paper we investigate the potential performance loss that may be incurred by using structured parallel programming models compared to unrestricted models. We present an analytical model that characterizes the performance loss in terms of application DAG *topology* and *workload distribution*. We validate the model for a large range of synthetic, and real-world DAGs, which are based on a number of well-known applications running on shared-memory and distributed-memory machines (Cray T3E, Origin 2000, Beowulf networks). All of our applications are in the dwarf application list recently selected by Berkeley researchers as key applications for parallel computing, which are proposed as an alternative to classical benchmarks in future research on parallel languages and computing [56].

The main conclusion of our study is that the performance loss due to choosing an SP-structured synchronization model, although theoretically unbounded, is quite small in real programs. Our experiments show that for applications that are properly load-balanced the loss is invariably limited by 10%. Even applications which carry the potential of high loss due to their

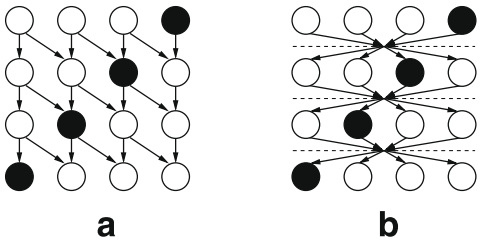

Fig. 1. NSP DAG (a) and SP DAG (b) of macro-pipeline.

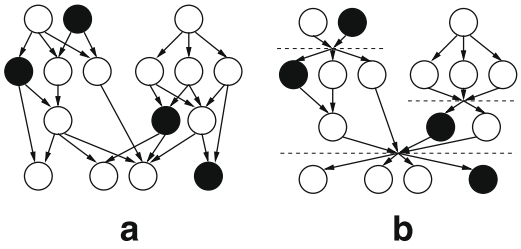

Fig. 2. NSP DAG (a) and SP DAG (b) of FEM computation.

programming structure benefit from a proper mapping, leading to negligible performance effects in practice. Only for applications that are executed with a very high degree of machine parallelism in combination with an extremely improbable (synthetic) workload distribution, the loss may increase in excess of a factor of 2. In all cases the loss is indicated by our model in terms of simple topological and workload distribution parameters. Given the very limited (or even zero) performance loss for the vast majority of applications, this theoretical and empirical evidence suggests a case in favor of using (or embedding) structured programming given their aforementioned, attractive properties. With this quantitative approach, this paper contributes towards further understanding of the complex trade-offs involved in choosing structured parallel programming models.

1.2. Related work

In preliminary work González et al. have investigated in the effect of using SP models on the performance of several applications (e.g. [26]). The current paper extends this earlier work, analyzing the entire taxonomy of application models and relating previous analytical results to more empirical data and real execution profiles. Some related work has also been performed. Work dedicated to comparing the expressive-power of LogP and BSP (examples of unstructured and structured models, respectively) appears in [6,46]. The comparison is based on asymptotic cross-simulation techniques, where it is assumed that any task may be interrupted when any other processor needs to communicate data. However, such a run-time model where a superstep (i.e., a barrier) is defined at arbitrary fine grain is unrealistic in practice, also in view of the significant performance penalty of introducing arbitrarily many synchronization barriers. Consequently, the work provides no information regarding the possible performance impact of using a restricted model in a practical setting, which would also need to take into account the specific impact of the application (e.g., its DAG).

A more quantitative study of different parallel programming techniques, including a nested-parallel BSP model, is given in [34]. However, the comparison includes a very limited set of applications with no extrapolation to the application characteristics which produce the effects.

Another study that bears some relation to our work is how to transform an NSP DAG into SP form with a minimum increase of the critical path when the exact workload distribution over the tasks is *known*. It has been conjectured that this loss may be bounded to be a very small constant c ($c = 2$ in [22], and $c = 4/3$ in [48]). However, in our case choosing between programming models is hardly done on a per-application basis, let alone based on task-level workload information, if available at all.

In summary, none of the above approaches consider the general problem of capturing the performance effects of using structured programming models in terms of basic application characteristics such as DAG topology and workload distribution, and we are not aware of any work that addresses this issue in a quantitative setting.

1.3. Paper outline

The paper is organized as follows. In Section 2 we formalize our main problem. In Section 3 we briefly discuss the algorithms used to transform actual NSP DAGs to the SP counterparts that we use as target in our performance comparison. In Section 4 we develop an approximate, analytical model of the performance loss which serves as the basis for the experimental sections that follow. In Section 5 we study the performance loss for various DAG topologies using synthetic workloads, while in Section 6 we study the performance loss for real-world workloads. In Section 7 we summarize our findings.

2. Formalization and rationale

In the following we formalize the problem, and develop the specific aims of our study.

Let $G = (V, E, \tau)$ denote a *weighted DAG* (from now on simply called DAG) where V is the finite set of nodes with cardinality $|V|$, E is the finite set of edges (or ordered node pairs) with cardinality $|E|$, and $\tau : V \to \mathbb{R}$ is the *workload distribution* function that assigns a weight (workload) to every node. Without loss of generality we assume each DAG to be single-rooted (possibly with zero workload).

A *path* $p(u, w)$ between two nodes u, w is the set of nodes $\{u, v_1, v_2, \ldots, v_n, w\}$ such that (u, v_1), $(v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, w) \in E$. The *depth level* of a node $d(v)$ is the maximum length of a path from the root to that node. The *maximum depth level* of a DAG $D(G)$, is the maximum length of a path from a root to a leaf. We say that a node v is *previous* to w, or w *depends on* v ($v \prec w$) if there is a path $p(v, w)$. We say that two nodes v, w are *connected* ($v \prec\succ w$) if v depends on w or w depends on v.

With respect to DAG topology we distinguish between regular and irregular DAGs. Regular DAGs are typically based on a repetition of a canonical substructure in both horizontal and vertical direction. The vertical regularity typically corresponds to sequential repetition of the computation (nodes organized in terms of *layers*, i.e., subset of nodes with equal depth), while horizontal regularity corresponds to computation (data) parallelism. A specific class of regular DAG is the *symmetric* DAG, which is defined as follows. Let $succ(v, k) = \{v' \in G(V, E) | \exists\ path(v, v') : depth(v') \geqslant depth(v) + k\}$ denote the successors of v after k node layers. In a symmetric DAG all nodes at depth $k + P$ are successors of each node at level k, where P is the parallelism of the DAG. That is, $\forall v \in G(V, E) : succ(v, P) = \{v' | depth(v') \geqslant depth(v) + P\}$. From the definition it follows that a

symmetrical DAG is layered and has regularity in both horizontal and vertical sense. A stencil DAG is an example of a symmetric DAG.

In the following we formulate the main objective of this paper. The *critical path length* $\varphi(G)$ is the maximum, accumulated load along the path, over all possible paths in G according to

$$\varphi = \max_{path \in G} \sum_{n \in path} \tau(n).$$

Let G_{NSP} denote the DAG associated with the original, possibly NSP structured, parallel computation. Let G_{SP} denote the DAG associated with an SP solution. Let

$$\gamma = \frac{\varphi(G_{SP})}{\varphi(G_{NSP})}$$

denote the performance loss due to the SP programming solution G_{SP} relative to the original computation G_{NSP}.

As described in the Introduction, γ is the primary aim of our study. In particular, we are interested in the *worst case* for γ (its *upper bound*), i.e., the greatest performance loss that a programmer can incur when choosing a convenient SP model instead of a more complex NSP model. In this way we establish a safe estimate of the potential performance damage when choosing an SP model.

With respect to our study on γ, a node in the DAG can model both computation and communication workload (and/or overhead). However, in our study on γ we will consider computations only. The reasons are twofold. First, when considering different programming models at program level one focuses on the (parallel) computation (algorithm), rather than the associated data transport, which is primarily determined by the subsequent machine mapping. Second, if one were to consider communication (nodes) as well, the results on γ will actually *decrease*, leading to overly optimistic results. This is due to the fact that both the NSP and SP model assume *asynchronous* communication, i.e., whenever a computation (node) has finished communication to successor nodes is initiated, independent of, e.g., barrier synchronization (typical of SP models, such as BSP). The effect of communication on γ can be modeled by extending the DAG with additional communication workload nodes for those edges that induce communication workload (e.g., between computation nodes that are to be mapped on different processors). As the mapping is not yet known the effect can only be accounted for by expanding each edge with a stochastic workload. As can be simply shown, this (as well as adding overhead terms) immediately decreases the upper bound on γ (confirmed by experiments in Section 6). Note that the decreasing effect on γ also occurs when considering contention effects (e.g., communication through busses) as the serialization occurs for both the NSP and SP DAGs.

Let $X : G \rightarrow G'$ denote a transformation of DAG G to G'. A transformation X that maps an NSP DAG into an SP DAG is called an *SP-ization technique*, which is the result of the effort by the programmer (or an algorithm) to express an NSP problem in terms of an SP programming model. The performance loss γ associated with the use of an SP model is a function of the initial DAG G_{NSP} and the applied transformation X. In theory, one would select (compute) the optimal X as to minimize the impact on γ. As in a usual programming situation the workload distribution is not known a priori (compiling and mapping is done a posteriori), knowledge on τ is not exploited in the transformation X. In general, X is clearly also a function of G_{NSP} topology (a programmer could apply different SP-izations for different G_{NSP} topologies). In our study, however, we will treat X and G orthogonally in the sense that we will only consider two generic (topology and workload independent) SP-izations, denoted X_1, and X_2, respectively, which are based on low-complexity algorithms, that correspond to the SP-izations typically performed by programmers. X_1 corresponds to the structure found in BSP models (non-nested parallelism), while X_2 corresponds to the more flexible, nested-parallel model (nested-BSP, Divide & Conquer, etc., see Section 3). As the specific characteristics of G_{NSP} in terms of τ and topology are not taken into account, X_i will generally be suboptimal. In that sense, γ is a pessimistic (upper) bound on the minimal (or zero) performance loss that could be theoretically achieved. Thus, in practice γ is likely to be smaller (i.e., better).

While γ represents the performance effects of choosing an SP programming model at the programming (or algorithmic) level, the actual effects at machine level must also be considered. Apart from γ, the actual execution time difference is also affected by the potentially positive effects of improved analyzability and schedulability on compilation and execution (as mentioned in Section 1), as well as by the potentially negative effects of increased implementation costs due to the additional synchronizations (barriers). Let $T(G)$ denote the *execution time* of a machine mapping of the computation corresponding to DAG G. Let

$$\Gamma = \frac{T(G_{SP})}{T(G_{NSP})}$$

denote the actual performance loss due to the use of an SP model. The difference between γ (potential performance loss, programming level) and Γ (actual performance loss, machine execution level) is illustrated by Fig. 3. The theoretical part of this paper focuses on γ, as Γ is affected by the mapping process, making it difficult to model. We will focus on Γ in the experimental study involving real programs. Although the study of Γ is important, as this is the actual performance loss as experienced by the programmer, Γ is typically much smaller than γ. The reason is that in many cases the mapping of the program onto the machine involves (1) a *reduction* of parallelism compared to the program-level DAG as many computation nodes are mapped onto the same processor (typically, the inherent parallelism at algorithm level is much higher than the number of

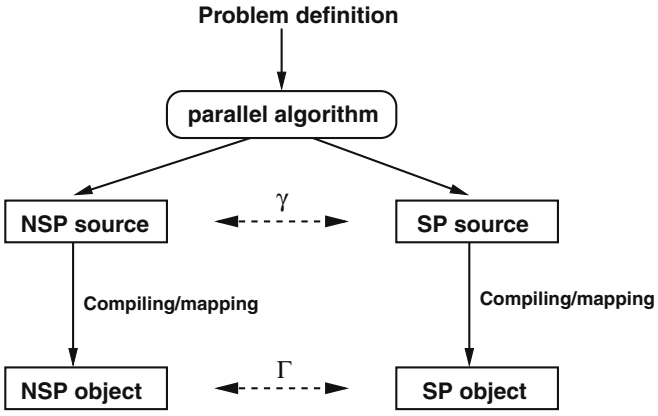

Fig. 3. Performance effects of choosing SP vs. NSP programming models.

processing resources available, which is usually limited to the order of thousands or less), (2) a *reduction* of the load imbalance in terms of τ, as mapping is aimed at load balancing, sometimes even reshaping and optimizing the synchronization structure to exploit the unstructured nature of the hardware and lower-level programming tools of the target platform.

As we will show in the paper, both a reduction of DAG parallelism, as well as load balancing, has a very favorable effect on γ (and hence Γ). Thus, studying γ for graphs which represent the structure of applications before mapping can be regarded as a safe (albeit somewhat pessimistic) estimate for the actual performance loss due to using structured models.

3. SP-ization algorithms

In this section we describe the two transformations X_1 and X_2 which we use in our study on γ (and Γ). Due to space limitations, we refer to [25] for a more elaborate presentation on SP-ization techniques. Both algorithms are based on characterizing a DAG in terms of node layers. X_1 is based on the familiar heuristic technique humans use to express parallel computations in an SP model as described in [42]. This heuristic is used, sometimes unconsciously, to program parallel algorithms with *bulk-synchronous* structures (see e.g. [24,30]). The transformation technique consists of adding the equivalent of a full barrier synchronization between each layer of nodes. The algorithm has time complexity $O(|E|)$.

X_2, a more sophisticated algorithm presented in [26], is also based on traversing the DAG downwards, layer by layer. When a new layer of nodes is traversed, nodes are grouped by their nearest common ancestor in the already transformed subDAG. Instead of using a full barrier synchronization, local barriers are issued for each group of nodes. This selective feature yields better results for irregular DAG topologies than X_1. The algorithm has time complexity $O(|E| + |V| \log |V|)$. An example of the result of applying X_1 and X_2 at the irregular FEM DAG shown earlier in Fig. 2, is shown in Fig. 4. Note that X_2 yields a better parallelization ($\gamma = 3$) compared to X_1 ($\gamma = 4$, again, the black nodes have unit load, the white nodes zero load, representing the worst case for γ). More details and correctness proofs can be found in [25].

As mentioned earlier, application of X_1 corresponds to the easy, BSP-like programming style, that will not always achieve equal parallelism compared to X_2. For regular topologies, however, X_2 does not often yield better results than X_1 as the topologies are usually not amenable to nested parallelization (e.g., stencils, pipelines). Hence, X_1 is most realistic when assessing γ (and Γ) for these problems. For irregular topologies, however, X_2 can yield better results (under specific circumstances as shown later on). Furthermore, application of X_1 is not so obvious due to the absence of a clearly layered structure. Hence, it is realistic to assume that nested parallelism will be exploited when using SP programming, and X_2 will be used.

4. Analytical model

In this section we develop an approximate, analytic model of γ to obtain basic insight into the influence of DAG topology and workload distribution. For analytic tractability reasons, we restrict ourselves to symmetric DAGs, and to stochastic node workloads that are independent, and identically distributed (iid).

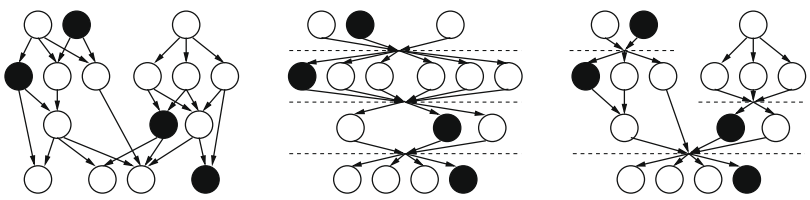

Fig. 4. Result of the application of X_1 (middle) and X_2 (right) to the example FEM NSP DAG (left), respectively.

Rather than selecting a negative-exponential distribution (or a variant), which is adopted in many performance modeling approaches for analytic tractability, we adopt a normal (Gaussian) distribution. The reason is that negative-exponential distributions (or distributions with comparable skewness and kurtosis) are rarely found in real workloads [21] due to the fact that at the task level workload distributions are primarily determined by the sum of many (smaller) terms such as conditional (data-dependent) control flow and contention for resources such as locks, memory, communication links, and CPUs in case of multithreading.

Despite the choice for a symmetric topology, the model captures the most salient factors with respect to topology and workload, that are to be varied in the computation-intensive, numerical experiments that follow later on in Section 5.

4.1. Model

Consider the DAG G_{NSP} associated with the symmetrical DAG shown in Fig. 5 (a 3-point stencil computation), along with G_{SP} due to SP-ization X_1 (X_2 yields the same result for this topology). In order to model γ we derive $\varphi(G_{NSP})$ and $\varphi(G_{SP})$. Let $\tau(v), v \in V$ be iid according to the normal distribution $N(\mu, \sigma)$ where μ denotes the mean workload, and σ^2 denotes the variance of the workload. Clearly, for deterministic, equal workloads ($\tau(v) = \mu, \sigma(v) = 0, \forall v \in V$), there would be no performance loss ($\gamma = 1$). Thus, workload variability (or imbalance) which is common in practice, is an important factor. Despite the fact that there is no exact closed-form solution for normal distributions, the mean critical path length φ for a single parallel section of P parallel nodes, iid according to $N(\mu, \sigma)$ can be analytically approximated [32] by

$$\varphi \approx \mu + \sigma\sqrt{2\log(0.4)P}.$$

For $P > 10$ the approximation error is less than 1% [21]. Let G_{NSP} (and hence G_{SP}) comprise D layers of P nodes each (D is depth). Then $\varphi(G_{SP})$ is approximated by

$$\varphi(G_{SP}) \approx D\left[\mu + \sigma\sqrt{2\log(0.4P)}\right].$$

While for SP DAGs the above estimation is straightforward (one of the advantages of SP DAGs), for NSP DAGs the situation is less straightforward. In order to estimate $\gamma(G_{NSP})$ we first approximate the NSP DAG by an SP DAG. Let $S = |E|/|V|$ denote the so-called *synchronization density* of G_{NSP}, i.e., the number of outgoing edges per node (for the 3-point stencil DAG $S = 3$). In [53,28] it is shown that for sufficiently large D the critical path length of a stencil-type NSP DAG of width P and synchronization density S approximately equals that of an SP DAG of same depth D and parallelism P' such that

$$P' \approx S + \log\frac{P}{2}.$$

In [53] it is empirically shown that this approximation is accurate for stencil-type DAGs within a small constant error (typically 10%). Hence, $\varphi(G_{NSP})$ can be approximated by

$$\varphi(G_{NSP}) \approx D\left[\mu + \sigma\sqrt{2\log(0.4(S + \log(P/2)))}\right].$$

Consequently, for sufficiently large D it holds

$$\gamma \approx \frac{\mu + \sigma\sqrt{2\log(0.4P)}}{\mu + \sigma\sqrt{2\log(0.4(S + \log(P/2)))}}. \tag{1}$$

4.2. Discussion

Despite the fact that the above model is an approximation that only applies to symmetrical DAGs, a number of basic properties of γ are uncovered, which, as shown later on, are much more general.

First of all, Eq. (1) indicates that γ increases (under)logarithmically with program parallelism. Similar to the macro pipeline example in Fig. 1 one can construct a "pathological" workload for which γ scales with P (shown by the dark nodes in Fig. 5). The logarithmic trend follows from the order statistics that underlie the model, by which it holds that increasing the number of edges (the barrier) yields logarithmic increase in critical path length (for iid workloads with non-zero

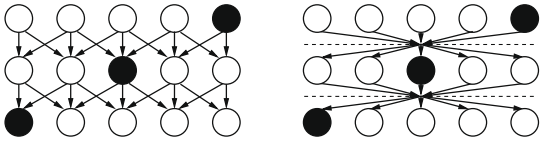

Fig. 5. NSP and SP version of 3-point stencil DAG.

variance). Consequently, DAG parallelism (P) is a key parameter, which indeed is confirmed by the numerical experiments presented later on, its particular region of interest clearly being the high range.

An equally important parameter is the variance in τ. The "pathological" example, mentioned earlier, is an example where σ is extreme from programming/load balancing point of view (either load values of 0 or 1). The order statistical model indicates that for large P (the region of interest) γ scales with σ. Again, this trend is confirmed by the numerical experiments presented later on.

The model also suggests that a third, important parameter is the synchronization density S. Clearly, for large S (e.g., $S = P$) γ will be 1 as each node in the NSP DAG has edges to each node in the next layer, which effectively constitutes a barrier. Hence, the inserted barrier in the SP DAG has no more impact (the NSP DAG is, in essence, already an SP DAG). For smaller S, however, the extent of this "virtual" synchronization barrier moves a number of layers down, i.e., the nodes downstream that are (indirectly) dependent on a node form a triangle with an apex angle that is linear in S. The number of nodes that can still execute in parallel (i.e., are outside a triangle) is therefore proportional to P/S. Consequently, the critical path length of the NSP DAG approaches that of an SP DAG with P/S parallel vertical subDAGs of width S. Hence, γ increases with decreasing S. For $S = 1$ we have P such subDAGs, each comprising a mere vertical string of nodes. Consequently, γ would be maximal (NSP DAG of parallelism P transformed into an SP DAG of parallelism P) but now with a barrier at each layer. Note, that in this sense the model is overly pessimistic since it is based on X_1, i.e., naively placing a barrier in between each layer. For $S = 1$, however, the NSP DAG happens to be already SP which is indeed recognized and exploited by X_2. Consequently, for X_2 we have $\gamma = 1$ rather than the maximum value (and to a lesser extent for $1 < S \leqslant 2$). Hence, we expect the maximum to be for small S, but larger than 1. Indeed, these phenomena are confirmed by the numerical experiments presented later on.

As explained in Section 2 we consider computation DAGs (topologies) only, as they provide the salient information on γ. Apart from not (explicitly) considering communication workload and/or overhead terms, the above model also does not explicitly consider the additional synchronization costs associated with SP models. The reason for this is the following. First, the communication workload aspects of synchronization are asynchronous (explained in Section 2), implying that including them will not change our findings. Furthermore, the typically logarithmic cost of implementing synchronization barriers (recursive doubling) is entirely subsumed by the logarithmic cost due to workload imbalance (i.e., the stochastic synchronization delays suffered by nodes that arrive at the barrier quicker than the last node) which is explicitly modeled by the order statistics model. Thus, explicitly modeling synchronization essentially does not affect the findings with respect to γ (but would increase the complexity of the DAGs and therefore the complexity of the analysis on γ). Hence, focusing on computation only does not affect the model.

In summary, although the model has been derived for symmetric DAGs, the theory suggests three key parameters, i.e., two topological parameters (P, S), and one workload parameter (σ, in the following we will assume $\mu = 1$ constant). The remaining experiments will be conducted in terms of these parameters to assess (1) the range of γ, and (2) to what extent these parameters predict γ.

5. Synthetic workloads

In this section we numerically investigate the behavior of γ for a wide range of NSP DAG topologies, and workload distributions, covering a vast range of application classes. (Note that we only focus on NSP DAGs as the much wider range of SP DAGs are irrelevant for our study.) The purpose of these experiments is to assess (1) what the influence of NSP DAG topology and workload is on γ, and (2) to what extent γ adheres to the analytical, three-parameter model.

The NSP topology spectrum includes regular DAGs (stencils, macro-pipelines, FFT, LU reduction), and irregular DAGs (random DAGs, iterative FEM solver, direct FEM solver). While most DAGs represent real applications, the random graphs are included in the study to represent highly unstructured applications and erratically designed parallel programs, which may not be covered by the classical application models. Note, that the above selection represents a very wide spectrum of topologies, ranging from regular to completely irregular, whose shape is controlled by user setpoints. Covering a large range of $|V|, P, S$ parameter values, our experiments involve over 1200 different DAG topologies.

In all cases, the workloads τ are synthetic, and are sampled from an $N(1, \sigma)$ distribution. The reason for selecting synthetic (random) workloads is that for the regular applications the task workloads are usually identical and highly deterministic. As in such cases γ is invariably close to unity (independent of topology) this would prohibit a sensible analysis of the influence of topology. Hence, we intentionally randomize τ (creating imbalance) to enable observing the effect of topology on γ. A second reason for using a stochastic workload is that this is a standard (performance modeling) approach to account for the effects of dynamic (data dependent) control flow and resource contention (scheduling). Combined with the wide range of topologies, the stochastic workload distributions (4 coefficient of variance settings, 25 samples per setting), the application space, ranging from highly regular DAGs to purely random DAGs, comprises more than 120,000 sample DAGs.

At this stage of our study we focus on γ instead of Γ. The experiments are partially intended to numerically validate the applicability of the analytic model, a process that would be needlessly complicated if the influence of the mapping process would also have to be taken into account (as in the case of Γ). Furthermore, as shown later on, applications which are well-programmed and properly mapped, do not exhibit the workload variability that is one of the parameters in such a study, which would lead to much too optimistic results regarding the worst case performance loss when using SP models. The effect of real workload on Γ is treated in Section 6.

5.1. Regular DAGs

In the following we study stencils, FFT, macro-pipeline, and LU reduction, all of which have distinct topological characteristics.

5.1.1. Stencil DAGs

As the analytical model was derived using a stencil topology, we first numerically study γ for a 3-point stencil DAG with parallelism P. With respect to the size $|V|$ we only consider P (width) rather than D (depth) as D cancels out (see Eq. (1)). Also from the triangle discussion in Section 4.2 it is clear that the minimum length of the DAG must be P/S in order to fully exploit the inherent parallelism in the NSP DAG. Experiments have verified that in practice D need not be larger than P/S to generate the worst case for γ. In the following, all experiments are based on DAGs that have sufficient depth to expose worst-case γ values. The experiments are based on averaging over 25 workloads sampled from $N(1,\sigma)$, while discarding negative values. From the NSP and the SP version the critical path length is computed, yielding γ. Four sets of experiments are performed where the parameter σ ranges over 0.1, 0.2, 0.5, 1.0.

Fig. 6 shows γ as function of P for the 3-point stencil DAG $(S = 3)$ with workload distribution parameter σ. The figure clearly shows the logarithmic effect of P and the large influence of σ. Initially, one might infer that using an SP model entails quite a performance penalty. However, the above values of P and σ should be interpreted with great care, as is discussed in Section 5.3.

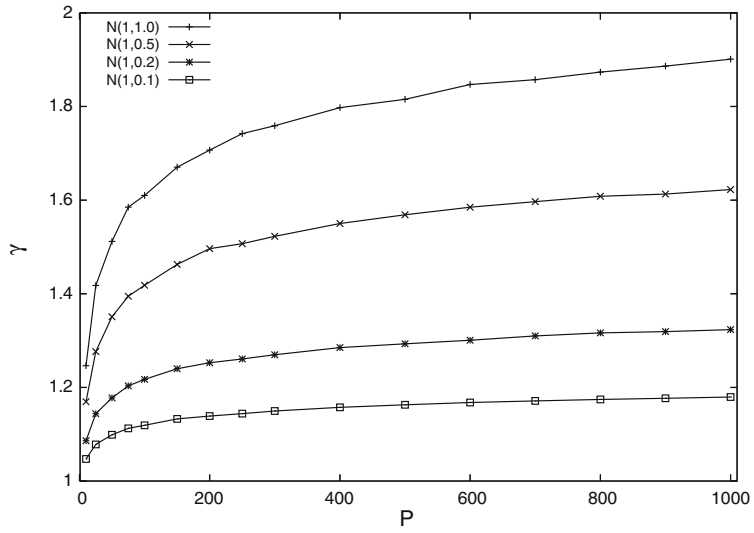

Fig. 6. γ vs. P for 3-point stencil.

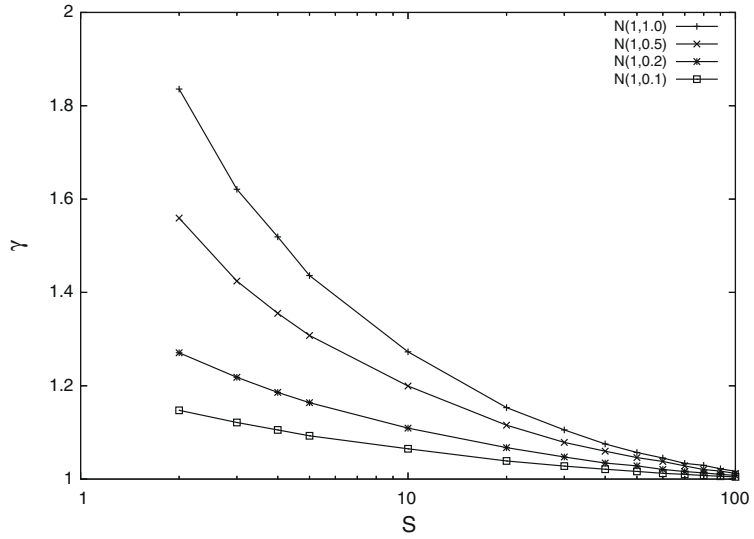

Fig. 7. γ vs. S for S-point stencil.

Fig. 7 shows γ as function of S for the stencil computation (i.e., an S-point stencil) with workload variability parameter σ while $P = 100$. The effect of S in the denominator of Eq. (1) is clearly visible, where $S = 100$ implies de facto full barrier synchronization (SP).

The above measurements are in agreement with the trends predicted by the model. The maximum for γ is reached for small $S(S \approx 2)$, while logarithmically increasing with P and increasing with σ. The latter two effects are directly caused by the statistical iid assumption, which, in terms of γ, is the most pessimistic assumption that can be made. For example, if we would assume that the workloads of a layer of nodes were fully *correlated*, then the order statistical effect would already be totally defeated, yielding $\gamma = 1$. Again, the model must therefore be seen as establishing a safe upper bound on the actual performance loss incurred in practice.

5.1.2. FFT

FFT is an example of a DAG that has a small synchronization density ($S = 2$). The FFT implements a Butterfly dependency structure with $\log P$ levels of depth [45], in which the distance to the neighboring node is doubled on each stage. The FFT DAG (and the SP version) are shown in Fig. 8, which also illustrates the worst case for γ using the familiar white/black workload distribution. Fig. 9 shows γ vs. D where D equals the number of butterfly stages. Since $P = 2^D$ the approximately linear plots are in agreement with the logarithmic trend seen earlier. As $S = 2$, in view of the earlier results, the γ values would be expected to be higher. However, the limited γ values are due to the fact that the depth D of the butterfly DAG is very small, compared to the reference model. While in the reference model γ is limited by P, rather than D, in the butterfly DAG the situation is reverse.

5.1.3. Macro pipeline

The macro pipeline is, again, an example of a DAG topology with a small S value ($S = 2$), which entails large γ values. Fig. 12 shows γ vs. P. Compared to the 3-point stencil DAG ($S = 2$) the γ values are consistently higher. This corresponds to the fact that the opportunities for parallelism in a P-wide pipeline (see Fig. 1) are greater than for, e.g., the 3-point stencil (see Fig. 5). In the pipeline case even P nodes can be issued a high (pathological) workload, compared to the $P/2$ nodes in the 3-point stencil, as illustrated by Fig. 1. Apart from the low S value, there is another reason for the higher γ values. Up to now the layered DAGs have been symmetric, and one parameter, S, has been shown to be sufficient to characterize these DAGs in relation to γ. In the pipeline case, the additional opportunities for parallelism are caused by the directional *bias* that the edges have (in the pipeline case all edges point down or to the right). In the biased case, higher values for D increase the

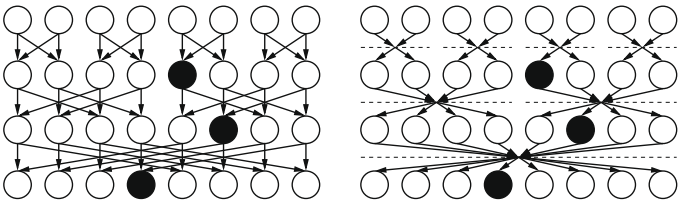

Fig. 8. FFT DAG (NSP and SP version).

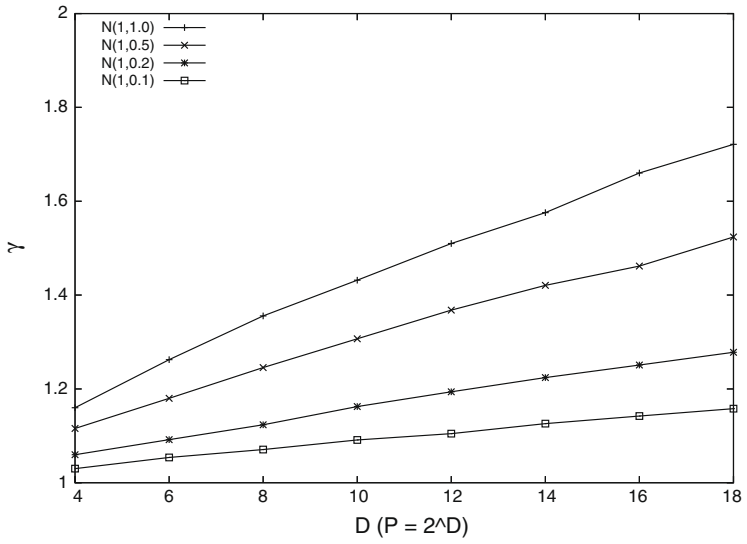

Fig. 9. γ vs. D for FFT.

opportunity for an unfavorable workload distribution, whereas for symmetric DAGs, D values larger than P/S do not offer more parallelism. Thus, although γ is still mainly determined by P, there is an additional effect for large D.

5.1.4. LU reduction

Another example of a biased DAG topology is the well-known LU factorization, of which the DAG is shown in Fig. 10. In contrast to the earlier DAGs, in the LU DAG the workload decreases with increasing row index. We accounted for this by proportionally decreasing μ and σ with increasing row index. Fig. 11 shows γ vs. P. Again, the γ values are somewhat higher compared to our reference model due to the (right) bias in the direction of all edges.

5.2. Irregular DAGs

While the analytical model is necessarily based on a DAG with regular topology, many application DAGs have less of a regular topology, reflecting, e.g., some form of dynamic or irregular application structure. In this section we study random topologies, and DAGs obtained from iterative and direct sparse-matrix (FEM) solvers. As explained in Section 3, for the irregular DAGs we use X_2.

5.2.1. Random DAGs

In this section we study γ for completely random DAGs. Unlike the layered topologies, which are generated in terms of P (and D), these DAGs are generated in terms of $|V|$ and S, according to the techniques presented in [2]. $|V|$ nodes are created while connecting node pairs (i,j) by an edge as determined by a Bernoulli trial with parameter p, given by $p = 2|V|S/(|V|(|V|-1))$, and only for $j > i$.

In order to study the effect of P and S we generate DAGs with $|V| = 32,\ldots,1024$, and subsequently measure the average P and S for each DAG ($P = 10,\ldots,380, S = 1.2,\ldots,P$). Fig. 13 shows γ vs. P for random DAGs with $S = 1.9,\ldots,2.1$ (i.e., the worst case γ values for S), again showing logarithmic behavior vs. P. Also the influence of σ essentially agrees with the analytical model. Due to the random synthesis, each point on a curve represents mean values of γ for hundreds of DAGs with widely varying topologies, demonstrating that the analytical model is not restricted to regular topologies.

Next, we study the impact of S, which is the other topology synthesis parameter next to $|V|$. Unlike the regular case, however, P and S are not independent. In order to eliminate the influence of P on S we rather study the impact of the size-*independent* parameter $R = S/|V|$ (i.e., $|E|/|V|^2$), rather than S. Fig. 14 shows γ vs. R involving DAG sizes ranging from $|V| = 32,\ldots,1024$ with an extremely large workload distribution variability ($\sigma = 1$). The curves are based on averaging over hundreds of DAGs per data point. The coinciding curves in the figure clearly show that R is a key parameter, while the size of

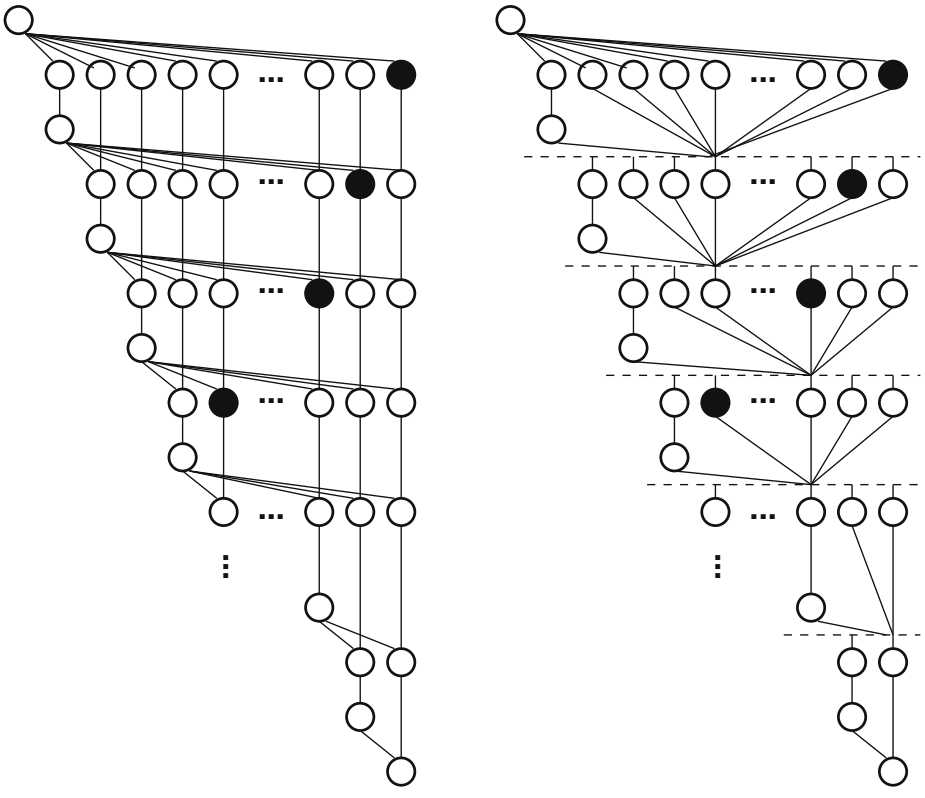

Fig. 10. LU DAG (NSP and SP version).

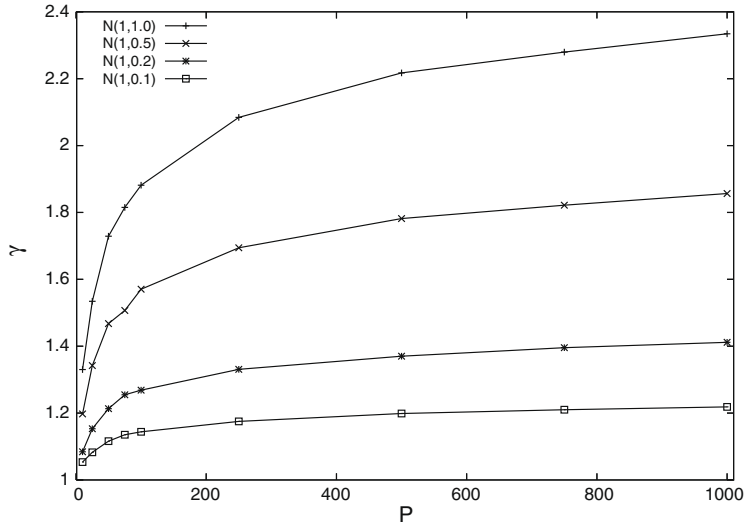

Fig. 11. γ vs. P for LU.

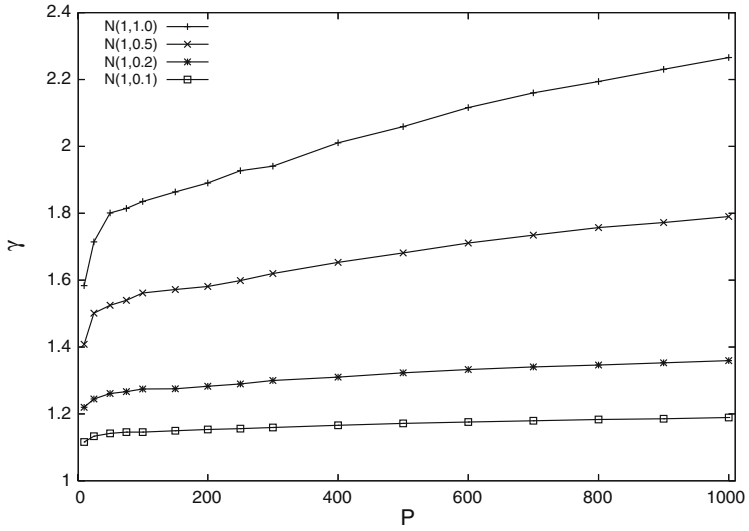

Fig. 12. γ vs. P for macro pipeline.

the DAG ($|V|$, and thus P) determines the maximum value of γ. The maximum is located around $R = 2/|V|$, which corresponds to $S = 2$. For $S > 2$ the effect of X_1 and X_2 are similar. For $S < 2$, however, X_2 is able to exploit the fact that the DAG is already close to SP (the DAG is SP for $S = 1$), whereas X_1 would introduce unnecessary global barriers.

5.2.2. Iterative FEM solvers

We have selected six data sets for an iterative FEM solver, obtained from the Everstine's collection within the Harwell–Boeing collection of sparse-matrices [16], available on the Matrix Market home page [43]. The six datasets represent civil engineering structures. They present different matrix-structure patterns and cover a wide range of matrix sizes ($87 \times 87, \ldots, 2680 \times 2680$). We have used METIS [35] (see, e.g., [49]) to partition the datasets for $P = 4, 8, 16, 32$, and 64 processors, yielding the (30) DAGs to be studied.

The DAG topologies are somewhat regular in the sense that they are layered where edges are restricted from one node layer to the next node layer (the DAG represents an iterative solver which implies that the edges repeat after each layer). The number of outgoing edges per node (S) is typically small, and has a small variance, usually exhibiting a region of spatial locality. Over the 30 DAGs the synchronization density varies between $S = 2.5$ and $S = 9.125$.

Fig. 15 shows γ vs. R for the 30 DAGs, averaging 25 workload samples for each variance value ($\sigma = 0.1, 0.2, 0.5, 1.0$). The results agree with the results for the random DAGs shown earlier (where $\sigma = 1.0$). Due to the fact that $S > 2$ for all topologies, we observe no drop yet in γ for the left-most DAGs. Note that the plots exhibit somewhat more sampling noise than the plots for the random DAGs which were averaged over much more samples.

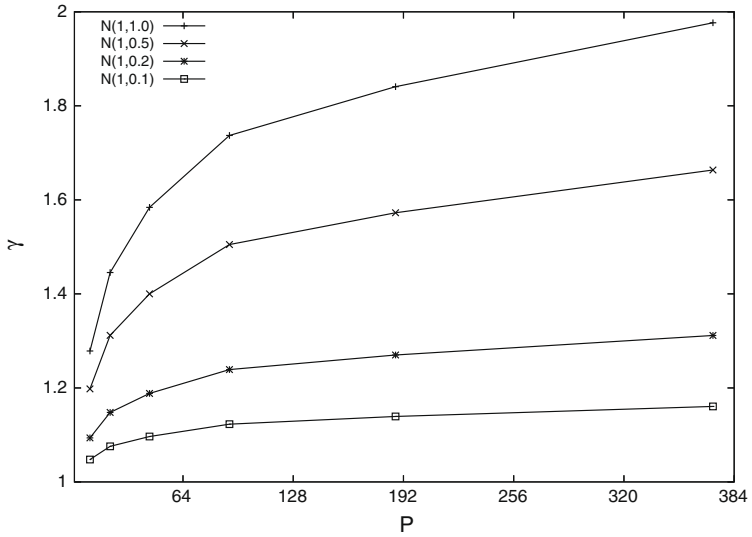

Fig. 13. γ vs. P for random DAGs.

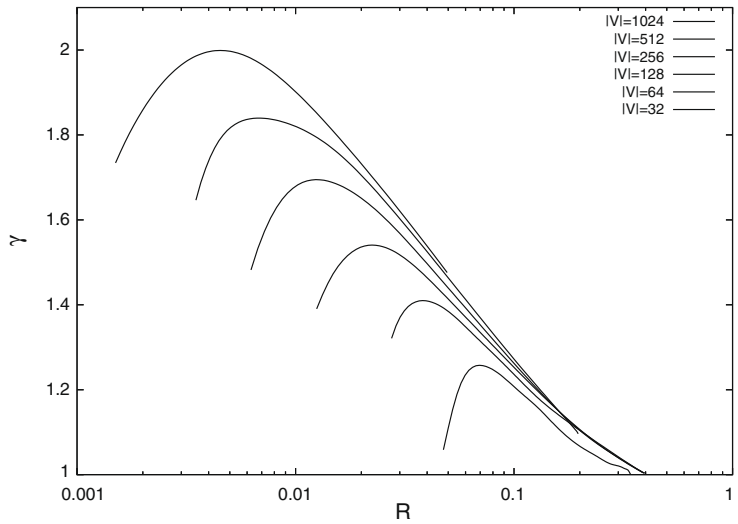

Fig. 14. γ vs. R for random DAGs.

5.2.3. Direct FEM solvers

We have selected six DAGs generated from DIANA [15], a direct FEM solver package which comprises software modules for domain decomposition and sparse-matrix factorization [38]. The six application DAGs are generated from the Diana domain decomposition module, and are subsequently executed in macro dataflow style.

The number of (coarse grain) nodes in the DAGs are $|V| = 58, \ldots, 3,541$, spanning a synchronization density range of $S = 2.8, \ldots, 13.8$. Compared to the iterative solver DAGs studied earlier, these DAGs have a more irregular topology, comprising an initial, highly parallel factorization phase, followed by a reduction phase, where the DAG ultimately converges into one end node (yielding the solution).

Fig. 16 shows γ vs. R for the six DAGs, averaging 25 workload samples for each variability value ($\sigma = 0.1, 0.2, 0.5, 1.0$). Again, the figure shows that γ essentially exhibits the same dependency on R as the irregular DAGs discussed earlier. The three DAGs with $S \approx 0.0035$ have somewhat different γ values due to limited parallelism, a concentration of workload in the upper layers, combined with sampling noise.

5.3. Summary

In the following we summarize our results. For the above measurements we conclude that for normally distributed, iid workloads, the behavior of γ is adequately captured by the analytical model. This implies that the model, although developed for regular stencils, applies to a vast range of regular, *and* irregular NSP DAG topologies, and is therefore much more fundamental. γ grows logarithmically with DAG parallelism, sharply decreases with synchronization density, and is highly

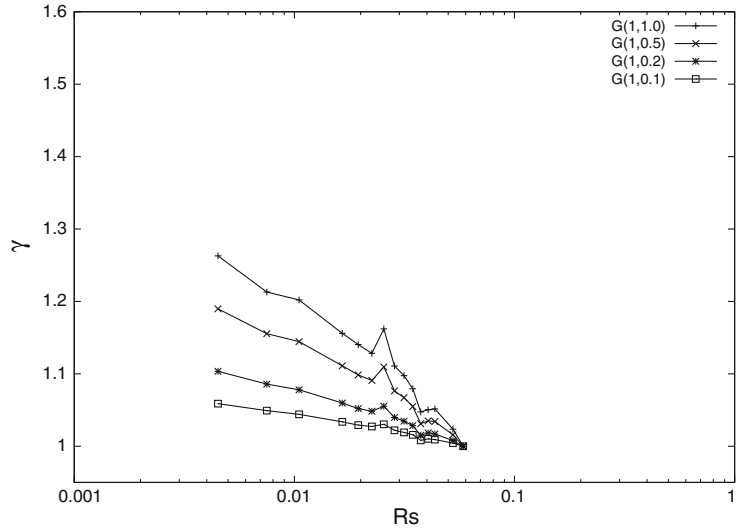

Fig. 15. γ for the six iterative solver DAGs.

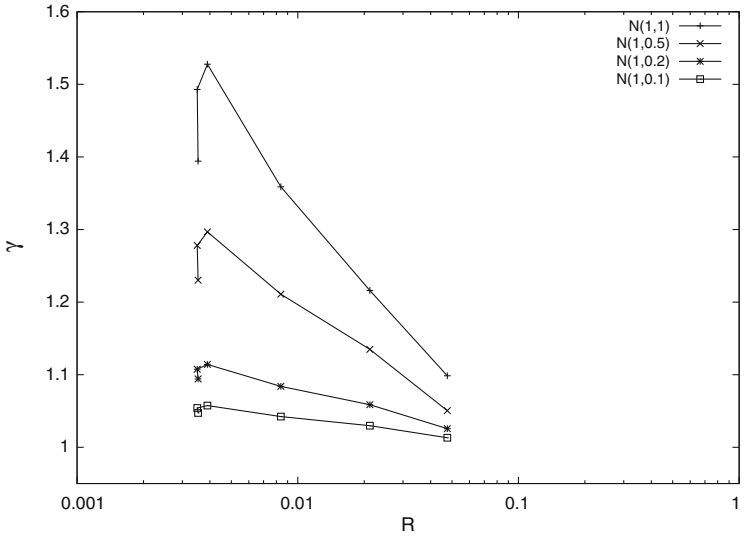

Fig. 16. γ for the six direct solver DAGs.

dependent on workload variance. For negligible variance, γ is invariably close to 1. Only for considerable variance (e.g., $\sigma = 1$) the γ numbers approach a factor of 2 for high degrees of DAG parallelism (e.g., $P = 1000$) and low synchronization densities (e.g., $S = 2$). Note, however, that this only applies when node workloads are independent. For highly correlated workloads, again, γ will quickly reduce to 1, as is shown in the next section.

6. Real workloads

In the previous section we numerically validated our theory on γ for a wide range of DAG topologies using synthetic workload distributions. To what extent the findings in the previous section apply in practice (in terms of Γ) highly depends on (1) the actual amount of parallelism at machine level, (2) actual workload distribution variance, and (3) task workload independence. As mentioned earlier, mapping the typically high levels of DAG parallelism to the lower levels of machine parallelism decreases P, but also σ as mapping aims to optimize workload balance.

In this section we present empirical results based on real workloads from regular and irregular computations. We determine Γ for the same range of applications studied in the previous section. The applications are expressed in terms of the original NSP version, and compared to an SP version based on X_1 for the regular DAGs and X_2 for the irregular DAGs, similar to Section 5.

In our experiments, we use MPI as programming interface. First we manually program a classical, optimized NSP version of each application in the study. Next, the SP version is derived from the NSP version by selectively programming additional

synchronization barriers. They represent the extra synchronizations caused by a nested-parallel construct used in a hypothetical structured language which would be systematically translated to the MPI interface in the same way. In this way we are sure that the rest of the program structure is identical and only the effect of the NSP to SP transformation is observed, factorizing out peculiarities of a particular SP or NSP compilation (mapping) process as much as possible [27].

6.1. Regular workloads

In this section we examine the regular applications studied in Section 5 (stencils, FFT, macro-pipeline, and LU). We have used the following three different machines: Cray T3E (3D grid architecture), Origin 2000 (CC-NUMA architecture), and Beowulf (NOW architecture). The DAGs are mapped onto P processors using a simple data partitioning in all cases, cyclic for LU to maintain load balance. While earlier we intentionally introduced stochastic workloads in order to investigate the influence of topology on γ, the real stencil, FFT, macro-pipeline, and LU computations inherently have very regular workloads where the variance between the tasks is virtually zero (tasks are identical, despite the fact that for LU the workloads decrease with increasing iteration, variance within one iteration is virtually zero). Hence, workload balance at the machine level is entirely determined by the load balancing qualities of the *mapping process* which are near-perfect. Hence, the issues related to Γ are essentially run-time effects such as the added cost of the additional (barrier) synchronizations.

Figs. 17–19 show the actual Γ measurements for the three machine architectures mentioned, respectively.

The small variations in Γ are due to execution time variance caused by link contention over the communication networks and stochastic operating system overhead. The figures clearly show that Γ is quite small. As P and σ are negligible, the performance loss is almost entirely due to the costs associated with the additional synchronization barriers.

6.2. Irregular workloads

In this section we examine the irregular applications studied in Section 5 (random DAGs, iterative FEM solvers, direct FEM solvers). The target machines are two heterogeneous Beowulf systems, one of 12 (University of La Laguna), and one of 40 nodes (University of Valladolid). To map the DAGs onto the P machine nodes, a simple, static list scheduler is used that creates a very good load balance as all applications are compute-bound (negligible communication, as mentioned earlier, communication only reduces the performance loss, yielding overly optimistic results on Γ).

6.2.1. Random programs

In this section we study the random DAGs described in Section 5. In contrast to the FEM solvers, which have application-specific (i.e., real) workloads, we apply synthetic workloads as the DAGs are synthetic. The workloads are sampled from a Gaussian distribution with μ sufficiently large enough to cause a CPU load that allows the execution to be compute-bound. The message-passing between the processors only implements the node precedence relations (the arcs in the DAG), i.e., empty messages. Inter-node data communication is not taken into account (as this does not change our findings with respect to Γ as explained earlier in the paper).

In contrast to the previous study on γ, we now deliberately focus on DAGs that have $S = 2$, representing the worst possible case, as shown in the previous study. Figs. 20 and 21 show Γ vs. P for 30,000 sample random DAGs with $|V| = 32$, and for 30,000 DAGs with $|V| = 1024$, respectively.

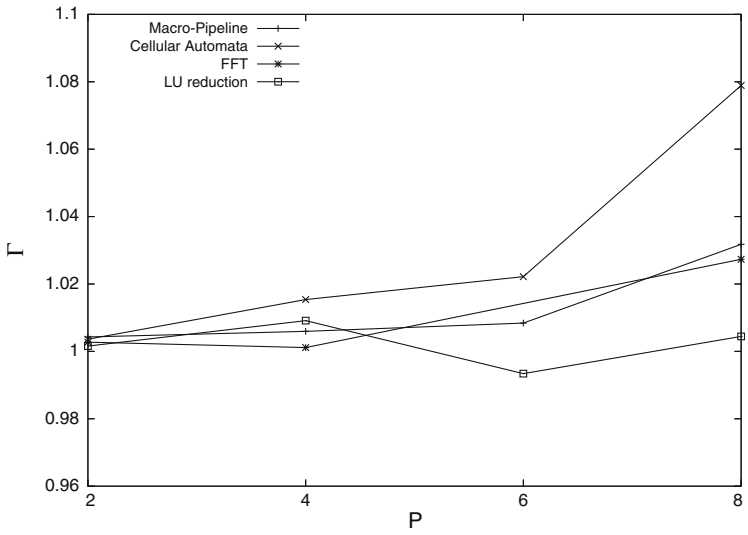

Fig. 17. Γ for regular DAGs on Origin.

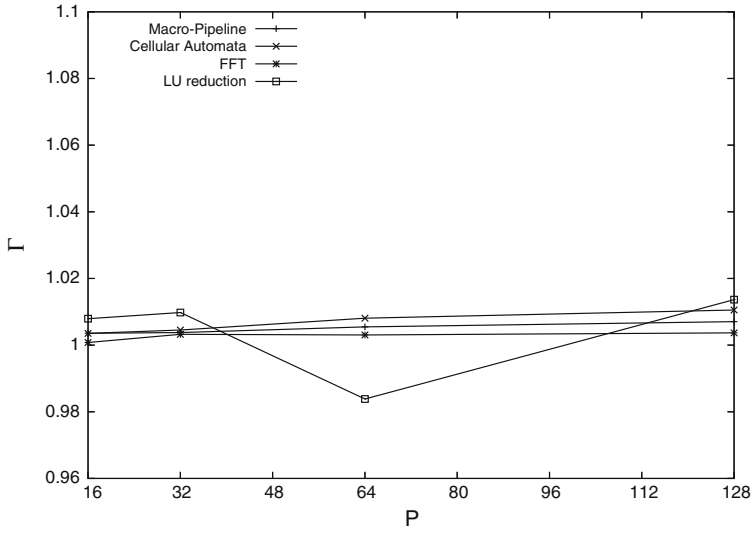

Fig. 18. Γ for regular DAGs on CrayT3E.

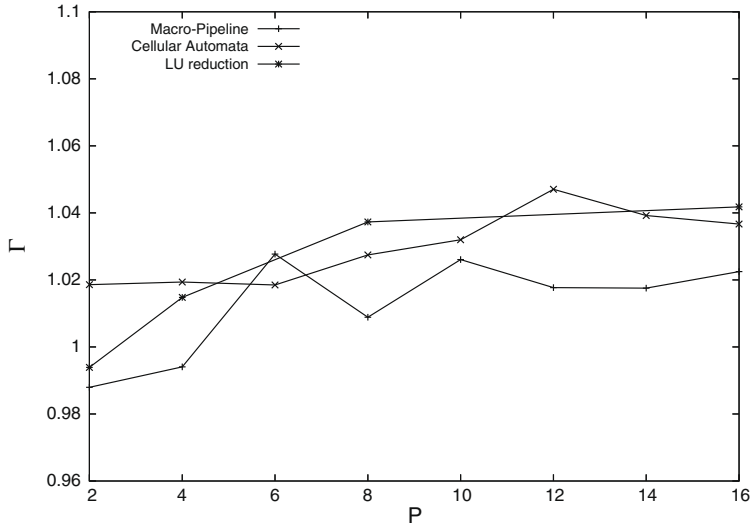

Fig. 19. Γ for regular DAGs on Beowulf.

The plots clearly show that Γ only slightly increases with P, reaching a maximum around $P = 2$ for $|V| = 32$, and around $P = 25$ for $|V| = 1,024$. The reason for the maximum of Γ is the limited parallelism in the DAG which roughly scales with $O\left(\sqrt{|V|}\right)$. For higher P actual parallelism does no longer increase while the higher degree of sequentialism in the mapped DAG actually reduces Γ. As mentioned earlier, all our experiments involve compute-bound DAG execution as including inter-node communication will only decrease Γ (and γ). To verify this, we have also executed the above random DAGs with a computation-to-communication-ration (CCR) of CCR = 1. In this case the maximum for Γ is less than 10%.

6.2.2. Iterative solvers

We use the same 30 DAGs as described earlier in Section 5 but now with the real workload. As the computational load is typically proportional to the number of nodes allocated per processor we can estimate μ and σ in terms of the number of nodes in each processor partition. In Table 1 we show the workload variability (σ/μ) measured for each DAG obtained by partitioning for a varying number of processors P. As the partitioning methods are designed to create a well-balanced partition, we observe very low workload variabilities. The only cases where the values are appreciable are found when the number of nodes per processor becomes small.

The above measurements suggest that Γ will be quite low. The potential impact for larger P (e.g., $P = 64$) is immediately defeated by the fact that for those ranges S is larger than 5. Although for large P workload variability is appreciable, the large correlation between tasks at the same DAG column (the vertical DAG axis is the iteration axis) prohibits any (logarithmic) growth of Γ.

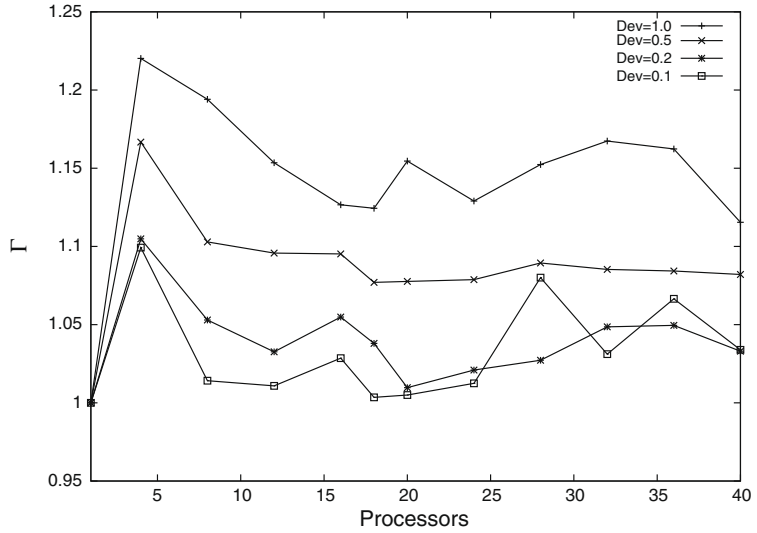

Fig. 20. Γ for random DAGs ($|V| = 32$).

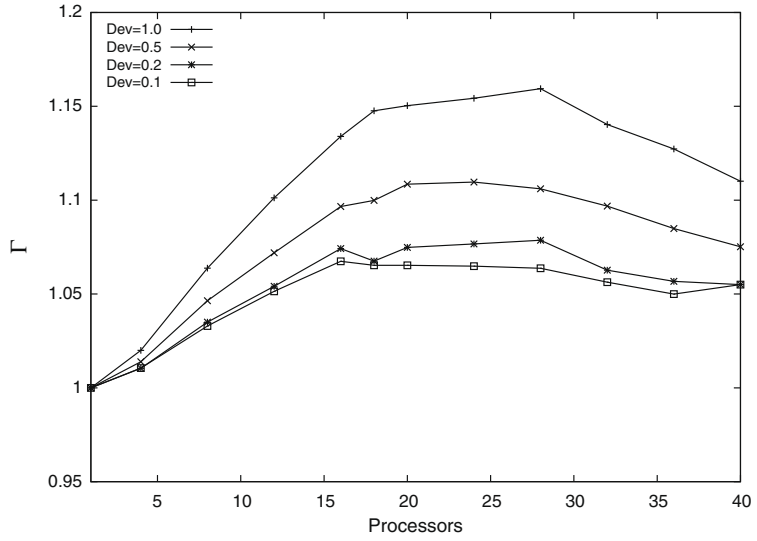

Fig. 21. Γ for random DAGs ($|V| = 1,024$).

Table 1
Estimated σ/μ for sparse iterative solver DAGs.

Problem	$\|V\|$	P				
		4	8	16	32	64
Tower	87	.01	.18	.21	.17	.57
Console	209	.03	.01	.15	.19	.15
Wankel	607	.05	.03	.04	.05	.33
Baseplate	1,005	.02	.02	.02	.04	.07
Sea chest	1,242	.02	.02	.02	.02	.07
Destroyer	2,680	.02	.02	.02	.02	.03

We have developed MPI skeleton programs which assign each part of the DAG to a different processor, annotating the *border sets* for each one. A border set is the collection of nodes which have a neighbor assigned to another given processor. Border-sets of a given processor are many times slightly overlapped, as there are some vertexes which have connections to several parts of the DAG. The program executes a loop in two stages: (a) computing the required number of floating point operations per assigned node; and (b) communicating new values from nodes in the border sets to the corresponding processors. The values for each border set are marshalled in a contiguous array. Thus, only one communication is issued from a

Table 2
Workload variability for sparse direct solver DAGs.

Problem	$\|V\|$	σ/μ
P59	59	2.10
P211	211	1.44
P772	772	7.11
P2014	2014	2.63
P3142	3142	5.28
P3541	3541	6.27

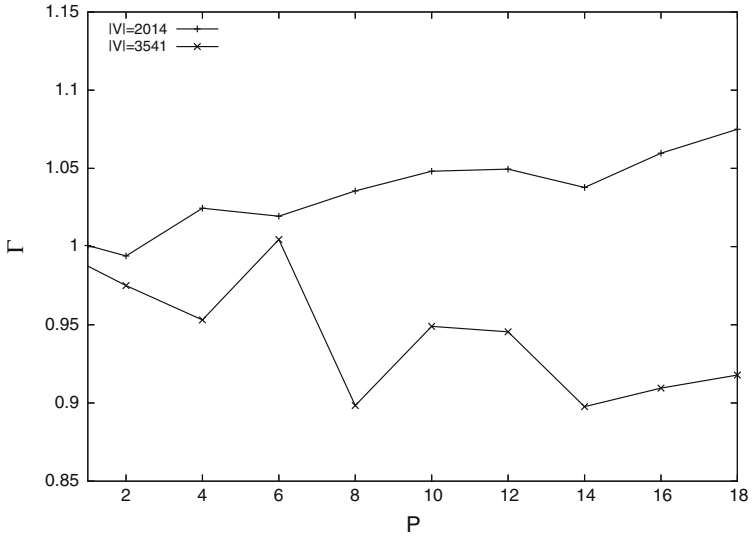

Fig. 22. Γ vs. P for two of the direct solver DAGs.

given processor to a neighbor on each iteration. In the SP version, a barrier is added after the communication stage, before the next loop iteration. This barrier would be implicit when using a native SP programming model.

We have tested executions from 100 to 1000 iterations of the loop to ensure that the results are independent of the number of iterations. As expected, in all cases Γ exhibits a mere 1% increase over unity. Hence, a visual presentation is omitted.

6.2.3. Direct solvers

Again, the study is based on the same six DAGs as in Section 5. Table 2 illustrates the high workload variability of the actual workload per node. However, since the DAGs are scheduled on a small(er) number of processors the effect of this high variability will be limited. The experiments are carried out for $P = 1, \ldots, 18$ processors. In almost all the cases this range covers the actual parallelism in the DAGs (the speedup for the largest DAG starts leveling off for $P > 20$).

Fig. 22 shows Γ vs. P for two of the six DAGs. The DAGs shown are the ones that have 3541 and 2014 nodes, respectively, and have been chosen because they yield the best and the worst case in terms of Γ, respectively, i.e., the Γ values of the other DAGs fall within both plots. These results imply that Γ is quite limited, and in some cases becomes even less than unity. This sharp contrast to the values that would be expected according to the variability numbers shown in Table 2 is due to the following reasons. First, this variability is mainly across task layers, which stratification is not affected by the SP-ization algorithm. Second, the results of *Gamma* less than 1 are produced because the simple scheduling technique used is not the most appropriate for the NSP DAGs. The performance of NSP DAGs could be improved by more sophisticated schedulers.

7. Conclusion

In this paper we studied the inherent impact of using structured parallel programming models on the amount of application parallelism that can be expressed when compared to programming without imposing any synchronization structure. In particular, we studied the impact for two well-known synchronization models, the simple BSP model, and the more versatile SP model, also known as Divide & Conquer, or nested parallel model. For a wide range of regular and irregular application DAGs with synthetic workloads we establish that the loss of parallelism, if any, is largely affected by DAG topology and workload distribution variability, and is characterized by a simple, analytic model. For stochastic, independent task workloads the loss of parallelism tends to grow logarithmically with DAG parallelism, the effect only being significant for small synchronization densities. Although theoretically unlimited, the loss of parallelism is quite limited, due to the fact that,

in practice, statistical workload variability and independence, as well as actual parallelism, is greatly limited in the mapping phase. For the entire range of regular and irregular applications running on various types of parallel machines we typically measure a performance penalty in the order of percents with occasional outliers of 10%. Whereas the merits in terms of, e.g., programming efficiency are generally hard to capture in quantitative terms, our study shows that imposing synchronization structure in parallel programming does not necessarily imply a significant performance sacrifice.

Acknowledgments

The authors extend their gratitude to Dr. Hai Xiang Lin of Delft University of Technology for making available the DIANA DAGs. We would like to acknowledge the support of the European Commission through IHP Grant Number HPRI-1999-CT-00026 (the TRACS program at EPCC) that gave us access to the Cray T3E and the Beowulf-1 machines. LaLaguna–Beowulf network was kindly made available by the University of La Laguna, partially supported by the EC FEDER Project and the Spanish MCyT (Plan Nacional de I+D+i, TIC2002-04498-C05-05, TIC2002-04400-C03 and TIN2007-62302). The Origin2000 machine and Lab-Beowulf Cluster were made available by the Computer Science Department, University of Valladolid, through CICYT Grant Contract IN97-65. Last but not least, we are very grateful for the reviewers' valuable comments, which greatly helped us to improve our paper.

References

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, J. Mellor-Crummey, S. Warren, C.-W. Tseng, Requirements for data-parallel programming environments, IEEE Parallel Distr. Technol., 1994, pp. 48–58.
[2] V.A.F. Almeida, I.M.M. Vasconcelos, J.N.C. Rabe, D.A. Menasc, Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems, in: Proceedings of Supercomputing'92, IEEE, Minn., MN, November 1992, pp. 683–691.
[3] M. Ãldinucci, A. B~enoit, Automatic mapping of ASSIST applications using process algebra, in: HLPP'05, 2005.
[4] G.R. Andrews, F.B. Schneider, Concepts and notations for concurrent programming, Comput. Surv. 15 (1) (1983) 3–43.
[5] G. Bilardi, Observations on universality and portability in high-performance computing, in: Proceedings of 1998 International Workshop on Innov. Archit., IEEE, 1999, pp. 21–26.
[6] G. Bilardi, K.T. Herley, A. Pietracaprina, BSP vs. Log *P*, in: Proceedings of 8th ACM Symposium on Par. Alg. and Arch. (SPAA'96), Padua, ACM, 1996, pp. 25–32.
[7] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, in: Proceedings of 5th PPoPP, ACM, 1995, pp. 207–216.
[8] R.D. Blumofe, C.E. Leiserson, Scheduling multithreaded computations by work stealing, in: Proceedings Annual Symposium on FoCS, November 1994, pp. 356–368.
[9] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, The Paderborn University BSP (PUB) library – design, implementation, and performance, in: Proceedings IPPS/SPDP'99, San Juan, Puerto Rico, Computer Society, IEEE, April 1999.
[10] D.R. Butenhof, Programming with POSIX(R) Threads, Addison-Wesley, 1997.
[11] R. Chandra, R. Menon, L. Dagum, D. Kohr, Parallel Programming in OpenMP, Morgan Kaufmann, 2000.
[12] M. Cole, Frame: an imperative coordination language for parallel programming, Technical Report EDI-INF-RR-0026, Div. Informatics, Univ. of Edinburgh, September 2000.
[13] The MPI Forum, MPI: a message passing interface, in: Proceedings of the conference on Supercomputing'93, ACM, 1993, pp. 878–883.
[14] J. Darlington, Y. Guo, H.W. To, J. Yang, Functional skeletons for parallel coordination, in: Europar'95, LNCS, 1995, pp. 55–69.
[15] DIANA FE Program, WWW, January 2000. <http://www.diana.tno.nl/>.
[16] I. Duff, R.G. Grimes, J.G. Lewis, Users' guide for the Harwell–Boeing sparse matrix collection (release i). Technical Report TR/PA/92/86, CERFACS, October 1992.
[17] T. El-Ghazawi, W. Carlson, T. Sterling, K. Yelick, UPC: Distributed Shared-memory Programming, Wiley-Interscience, 2003.
[18] L. Finta, Z. Liu, I. Milis, E. Bampis, Scheduling UET–UCT series–parallel graphs on two processors, Theoret. Comput. Sci. 162 (August) (1996) 323–340.
[19] I.T. Foster, K. Mani Chandy, Fortran M: a language for modular parallel programming, J. Parallel Distr. Comput. 26 (1995) 24–35.
[20] K.S. Gatlin, Trials and tribulations of debugging concurrency, ACM Queue 2 (7) (2004) 67–73.
[21] H. Gautama, A.J.C. van Gemund, Low-cost static performance prediction of stochastic parallel task compositions, IEEE Trans. Parallel Distr. Syst. 17 (1) (2006) 78–91.
[22] A.J.C. van Gemund, The importance of synchronization structure in parallel program optimization, in: Proceedings of 11th ACM ICS, Vienna, July 1997, pp. 164–171.
[23] A.J.C. van Gemund, Symbolic performance modeling of parallel systems, IEEE Trans. Parallel Distr. Syst. 14 (2) (2003) 154–165.
[24] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, J. Parallel Distr. Comput. 22 (2) (1994) 251–267.
[25] A. González-Escribano, Synchronization Architecture in Parallel Programming Models, PhD Thesis, Dpto. Informtica, University of Valladolid, July 2003.
[26] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, Mapping unstructured applications into nested parallelism, in: High Perf. Comp. for Comput. Science (Selected Papers), LNCS 2565, Springer, 2003, pp. 407–420.
[27] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, Tools to schedule and simulate parallel computations expressed as directed graphs. Technical Report IT-DI-2007-0001, Dpto. Infomática, Universidad de Valladolid, April 2007.
[28] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, H.-X. Lin, V. Vaca-Díez, Expressiveness versus optimizability in coordinating parallelism, in: Parallel Computing, Fundamentals & Applications, Proc. Int. Conf. ParCo99, Delft (The Netherlands), Imperial College Press, August 1999, pp. 526–533.
[29] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, R. Portales-Fernández, J.A. Caminero-Granja, A preliminary nested-parallel framework to efficiently implement scientific applications, in: VECPAR 2004, LNCS 3402, Springer, April 2005, pp. 541–555.
[30] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, Portable and efficient parallel computing using the BSP model, IEEE Trans. Comput. 48 (7) (1999) 670–689.
[31] T. Gross, D.R. O'Hallaron, J. Subhlok, Task parallelism in a high-performance Fortan framework, IEEE Parallel Distr. Technol. (1994) 16–26.
[32] E.J. Gumbel, Statistical Theory of Extreme Values (Main Results), Wiley Publications in Statistics, John Wiley & Sons, 1962. pp. 56–93 [Chapter 6].
[33] P.G. Joisha, P. Banerjee. PARADIGM(version 2.0): a new HPF compilation system, in: Proceedings IPPS/SPDP'99, IEEE Computer Society, San Juan, April 1999.
[34] B.H.H. Juurlink, H.A.G. Wijshoff, A quantitative comparison of parallel computation models, ACM Trans. Comput. Syst. 16 (3) (1998) 271–318.
[35] G. Karypis, METIS: family of multilevel partitioning algorithms, WWW, 2002. <http://www-users.cs.umn.edu/~karypis/metis/>.
[36] C.W. Kessler, NestStep: nested parallelism and virtual shared memory for the BSP model, in: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas (USA), June–July 1999.

[37] E.A. Lee, The problem with threads, Computer 39 (5) (2006) 33–41.
[38] H.X. Lin, A general approach for parallelizing the FEM software package DIANA, in: Proceedings of High Performance Computing Conference'94, National Supercomputing Research Center, National University of Singapore, 1994, pp. 229–236.
[39] K. Lodaya, P. Weil, A Kleene iteration for parallelism, in: Proceedings of FST & TCS'98, LNCS 1530, Springer, 1998, pp. 355–366.
[40] K. Lodaya, P. Weil, Series–parallel posets: algebra, automata, and languages, in: Proceedings of STACS'98, LNCS, vol. 1373, Springer, Paris, 1998, pp. 555–565.
[41] K. Lodaya, P. Weil, Series–parallel languages and the bounded-width property, Theoret. Comput. Sci. 237 (2000) 347–380.
[42] A.D. Malony, V. Mertsiotakis, A. Quick, Automatic scalability analysis of parallel programs based on modeling techniques, in: Comp. Perf. Eval.: Modeling Techniques and Tools (LNCS 794), Springer, Berlin, May 1994, pp. 139–158.
[43] National Institute of Standards and Technology (NIST), Matrix Market, WWW, 2002. <http://math.nist.gov/MatrixMarket/>.
[44] R.V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, H.E. Bal, Satin: simple and efficient Java-based grid programming, J. Parallel Distr. Comput. Prac. (2004).
[45] M.J. Quinn, Parallel Computing: Theory and Practice, McGraw-Hill, 1993.
[46] V. Ramachandran, B. Grayson, M. Dahlin, Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design, in: Proceedings of ACM–SIAM SODA'99, 1999, pp. 957–958.
[47] R.A. Sahner, K.S. Trivedi, Performance and reliability analysis using directed acyclic graphs, IEEE Trans. Softw. Eng. 13 (10) (1987) 1105–1114.
[48] A.Z. Salamon, Task Graph Performance Bounds Through Comparison Methods, MSc Thesis, Fac. of Science, Univ. of the Witwatersrand, Johannesburg, January 2001.
[49] K. Schloegel, G. Karypis, V. Kumar, CRPC Parallel Computing Handbook, Graph Partitioning for High Performance Scientific Simulations, Morgan Kaufmann, 2000.
[50] D.B. Skillicorn, A cost calculus for parallel functional programming, J. Parallel Distr. Comput. 28 (1995) 65–83.
[51] D.B. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Comput. Surv. 30 (2) (1998) 123–169.
[52] K. Takamizawa, T. Nishizeki, N. Saito, Linear-time computability of combinatorial problems on series–parallel graphs, J. ACM 29 (3) (1982) 623–641.
[53] A. Vaca-Díez, Tools and techniques to assess the loss of parallelism when imposing synchronization structure, Techn. Rep. 1-68340-28(1999)02, TU Delft, 1999.
[54] J. Valdés, R.E. Tarjan, E.L. Lawler, The recognition of series–parallel digraphs, SIAM J. Comput. 11 (2) (1982) 298–313.
[55] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[56] WWW, The landscape of parallel computing research: a view from berkeley, WWW, November 2006. <http://view.eecs.berkeley.edu/>.