A simulator for adaptive parallel applications

Basile Schaeli, Sebastian Gerlach, Roger D. Hersch

Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

School of Computer and Communication Sciences

{basile.schaeli, sebastian.gerlach, rd.hersch}@epfl.ch

Abstract

Dynamically allocating computing nodes to parallel applications is a promising technique for improving the utilization of cluster resources. Detailed simulations can help identify allocation strategies and problem decomposition parameters that increase the efficiency of parallel applications. We describe a simulation framework supporting dynamic node allocation which, given a simple cluster model, predicts the running time of parallel applications taking CPU and network sharing into account. Simulations can be carried out without needing to modify the application code. Thanks to partial direct execution, simulation times and memory requirements are reduced. In partial direct execution simulations, the application's parallel behavior is retrieved via direct execution, and the duration of individual operations is obtained from a performance prediction model or from prior measurements. Simulations may then vary cluster model parameters, operation durations and problem decomposition parameters to analyze their impact on the application performance and identify the limiting factors. We implemented the proposed techniques by adding direct execution simulation capabilities to the Dynamic Parallel Schedules parallelization framework. We introduce the concept of dynamic efficiency to express the resource utilization efficiency as a function of time. We verify the accuracy of our simulator by comparing the effective running time, respectively the dynamic efficiency, of parallel program executions with the running time, re-

Computer System Sciences, Vol. 74, Issue 6, 2008, pp 983-999

spectively the dynamic efficiency, predicted by the simulator under different parallelization and dynamic node allocation strategies.

Keywords: Adaptive parallel application simulation, performance prediction, dynamic efficiency, sensitivity analysis, partial direct execution.

1. Introduction

Recent studies show that many parallel applications do not fully use the available hardware [7], [12]. Although some applications are inherently difficult to parallelize efficiently, many other applications could be improved by using better parallelization strategies and problem decomposition parameters. Moreover, most parallel job scheduling systems allocate a constant number of compute nodes to an application, causing nodes to become idle or underutilized when the application's processing power requirements vary over the course of execution. Adapting the allocation of nodes to the applications' computation needs may thus further increase the utilization of computing resources during program execution.

The choice of an efficient problem decomposition may depend on the input data of the application, as well as on the number of available nodes. Similarly, taking good decisions about how and when to modify the allocation of compute nodes requires a priori knowledge about the *dynamic efficiency* of the application, i.e. its utilization of computation resources as a function of time. Many test runs must therefore be performed to obtain the necessary information. This testing phase can be time consuming on busy production parallel systems, since jobs must wait until processing time becomes available. Being able to use a desktop computer to produce detailed simulations and provide information about the dynamic efficiency as well as the effectiveness of the chosen problem decomposition can therefore reduce the time and cost of parallel application development.

This paper describes the simulation capabilities that have been integrated into the Dynamic Parallel Schedules (DPS) parallelization framework [9]. The integration of the simulator within the framework enables simulating a parallel application by fully or partially executing the application code. This enables

reconstructing its exact behavior. Since the simulator also executes the DPS runtime code, features such as the dynamic allocation of processing nodes or the production of an execution trace are also simulated.

The problem of dynamically allocating resources to parallel applications has been previously considered [6], [13], [20]. However, according to our knowledge this paper presents the first simulator that predicts the performance of real adaptive applications, i.e. applications whose mapping to computation nodes may vary over time during program execution.

Much research has already been carried out on predicting the performance of parallel programs with static node allocation. Purely analytical models are generally tailored to a specific application [14] or to a class of parallel programs, such as fork-join applications [17]. Other models have two levels of hierarchy [1], with a higher-level component representing the task-level behavior of the program and a lower-level component representing individual task execution times. These models describe the task-level behavior as a task graph [1], [16] or as a timed Petri net [3]. Approaches for modeling individual task execution times include measurements [3], [14], stochastic models [16], [17] and the association of an application signature and a machine profile [19].

MPI-SIM [18] and its extension COMPASS [4] are two simulators that predict the performance of MPI programs by executing the actual application code. The simulation functionality is provided by a modified library that implements the most common MPI calls. Both MPI-SIM and COMPASS derive computation times through direct execution [7], i.e. by executing and measuring the running time of the application code. The simulation should therefore run on the same hardware as the parallel application. The code does not need to be modified, and no distinct model of the application must be maintained. However, a single processor performs all computations and the whole problem must fit into the memory of a single computing node, thus limiting the size of applications that can be simulated. MPI-SIM and COMPASS alleviate these problems through parallel simulation, which however requires the parallel system to be available.

We follow a mixed approach, where the task-level behavior is obtained by executing the runtime and application code within the simulator. However, computations that have no impact on the task-level behavior of the application may be replaced by duration estimates. Additionally, we may reduce memory

usage by avoiding data structure allocations. The direct execution drawbacks are therefore considerably reduced. We refer to this mixed approach as *partial direct execution*.

Unlike other simulators which ignore network delays [2], [17], we take network overheads into account by using a simple model and a small set of platform-specific parameters. As a result, our simulator is portable and the execution of parallel programs can be accurately simulated on a desktop computer.

Identifying platform parameters and task duration estimates enables simulations to provide insights about the sensitivity of the application to each parameter. This helps identifying potential performance optimizations as well as determining whether the execution is CPU- or network-bound. Simulations therefore enable application developers to study and improve the performance of their applications without maintaining a separate model and without having access to a parallel machine.

The paper is organized as follows. Section 2 briefly describes the Dynamic Parallel Schedules parallelization framework, and Section 3 explains the integration of the simulator within DPS. The assumptions made about the parallel system are described in Section 4. We show simulator validation results for an LU factorization application in Section 5, and for a load-balanced traveling salesman problem in Section 6. Sections 7 and 8 respectively show the benefits of the partial direct execution and a detailed sensitivity analysis of the LU factorization application. This sensitivity analysis provides insight about the behavior of the application for different cluster model parameters. Section 9 draws the conclusions.

2. The Dynamic Parallel Schedules framework

DPS [9] describes a distributed memory parallel computation as a *flow graph* composed of serial operations arranged to form an acyclic directed graph, whose edges are defined by the messages that transit between operations. The flow graph describes the asynchronous flow of data between operations.

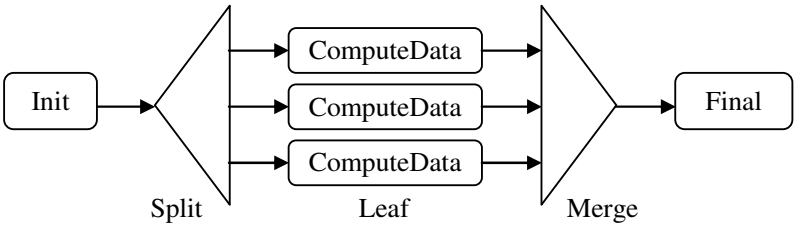

Figure 1. Flow graph describing a high level task divided into subtasks by a custom split operation. Leaf operations perform their tasks in parallel.

The particular implementation of operations is left to the programmer, but each operation must be of one of four fundamental types: *leaf*, *split*, *merge* or *stream*. *Leaf* operations accept a single input and generate a single output message. *Split* operations take one input message and generate one or several output messages. *Merge* operations expect one or several input messages, and generate a single output message once all expected input messages have been received. Split operations are typically used to subdivide a high-level task into several subtasks that can be performed in parallel. Computation results are then collected and aggregated by the matching merge operation (Figure 1). The fourth operation type, the *stream*, places no restriction on the number of input and output messages. It allows the programmer to refine the synchronization granularity by allowing new messages to be streamed out as soon as specific groups of incoming messages have been received.

Operations within a flow graph are carried out within threads. Each thread is wrapped within a data structure that provides an execution environment for a set of operations, and queues incoming messages until they are processed. Messages are transferred as soon as they are generated, making the execution of DPS applications fully pipelined and asynchronous, with automatic overlapping of communications and computations. In order to avoid overflowing reception queues, a flow control mechanism can be used to limit the number of messages in circulation between a split operation and the matching merge operation.

Leaf operations are executed atomically. Other operations may be suspended during their execution, e.g. due to the flow control mechanism or when merge and stream operations wait for messages that did not yet arrive. The suspension prevents deadlocks by allowing other operations to run.

The deployment of a DPS application is performed dynamically, and relies on a remote launching mechanism to create new application instances as needed. In each application instance, a *thread manager* handles thread creation and destruction requests. Threads can be migrated by transferring the corresponding data structures to another application instance [10]. A communication layer, based on TCP sockets, hides network transfers and physical thread location from the application programmer.

3. Structure of the simulation system

Most of the information needed to reconstruct the execution of a parallel application is only available at runtime. The execution pattern may for instance be data dependent, and intermediate computation results may influence future data distribution decisions. In addition, parallel programs may implement load-balancing schemes that make it very difficult to predict the location of computations and the resulting network transfer patterns. This motivated our decision to integrate the simulation capabilities within the DPS parallelization framework. By directly executing code both from the application and from the framework runtime, the simulator knows the destination of every message, the number of messages sent by each split operation and the current number of processing nodes and threads. An application is simulated by simply activating a compilation flag.

In order to emulate the deployment of threads onto compute nodes, the simulator uses a modified remote launching mechanism that instantiates a new thread manager for each application instance that would have been launched in a real execution. It simultaneously maintains a virtual representation of each computing node on which the application is deployed (Figure 2). The TCP network layer is replaced by a simulated network layer, which handles all communications between the virtual nodes. All mechanisms that rely on the network layer, such as the transfer of messages or the dynamic allocation of threads, are used without modifications within simulated applications.

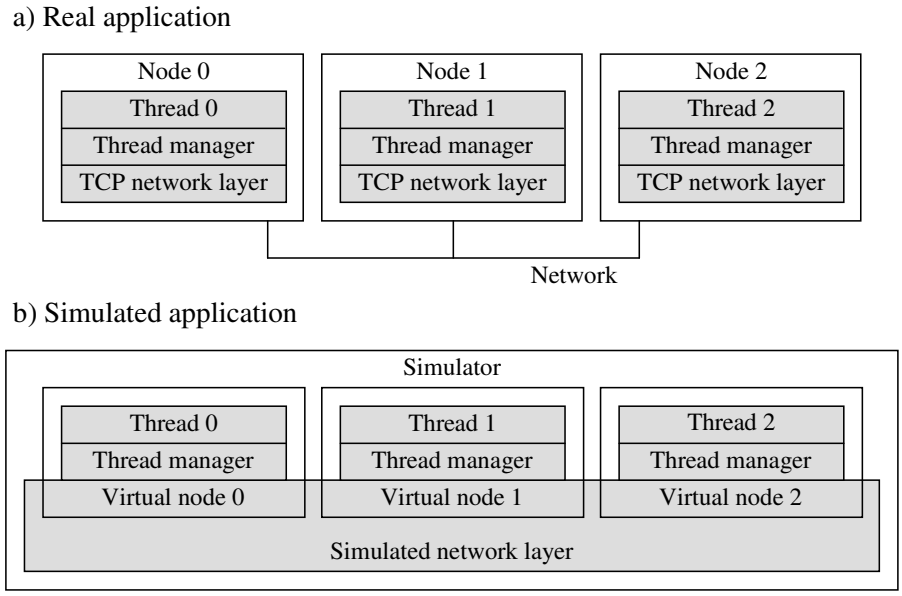

Figure 2. Allocation of threads to computing nodes in a real and simulated application (more than one thread may be running on each node). For the simulation, every thread manager is attached to a virtual node.

The simulator reconstructs the application execution by keeping track of which threads and which virtual nodes execute the different operations. Since operations may be suspended during their execution, the simulator subdivides them into *atomic steps*, i.e. operation parts which execute without being suspended. Message transfers are also assimilated to atomic steps. Except for the first atomic step of a flow graph, an atomic step starts when another atomic step terminates, and ends when a message transfer completes or when an operation suspends or finishes its execution.

The simulator code runs within its own thread, called the *simulator thread*. Threads that execute DPS operations are referred to as *computation threads*. The simulator thread maintains a simulation clock and controls the activation of the computation threads, ensuring that no two threads run simultaneously. When a computation thread completes the execution of an atomic step, it queues the atomic step and its duration within the simulator. The computation thread then suspends its execution and resumes the simulator thread. When the simulator thread is running, it advances its simulation clock to the point where an atomic step completes. If the completed atomic step represents a message transfer, the simulator resumes

the computation thread that receives the transferred message. If the atomic step belongs to an operation, the simulator resumes the computation thread running that operation. In all cases the simulator thread is suspended while the computation thread is running (Figure 3).

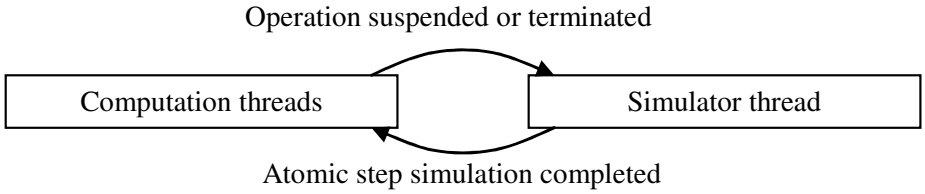

Figure 3. Alternating execution of DPS operations and of the simulator.

Figure 4 shows the atomic steps of the execution of a simple flow graph deployed on 3 nodes as in Figure 2. One node runs the operations *Split* and *Merge*, while the other two run the leaf operations *Leaf1* and *Leaf2*. The split operation is composed of the atomic steps S1 and S2, which respectively generate the message transfers T1 and T2. Each leaf operation consists of a single atomic step (L1 and L2). The subsequent message transfers T1' and T2' trigger the execution of the atomic steps M1 and M2 within the operation *Merge*. The gap between M1 and M2 indicates that the *Merge* operation is suspended while waiting for the message from L2.

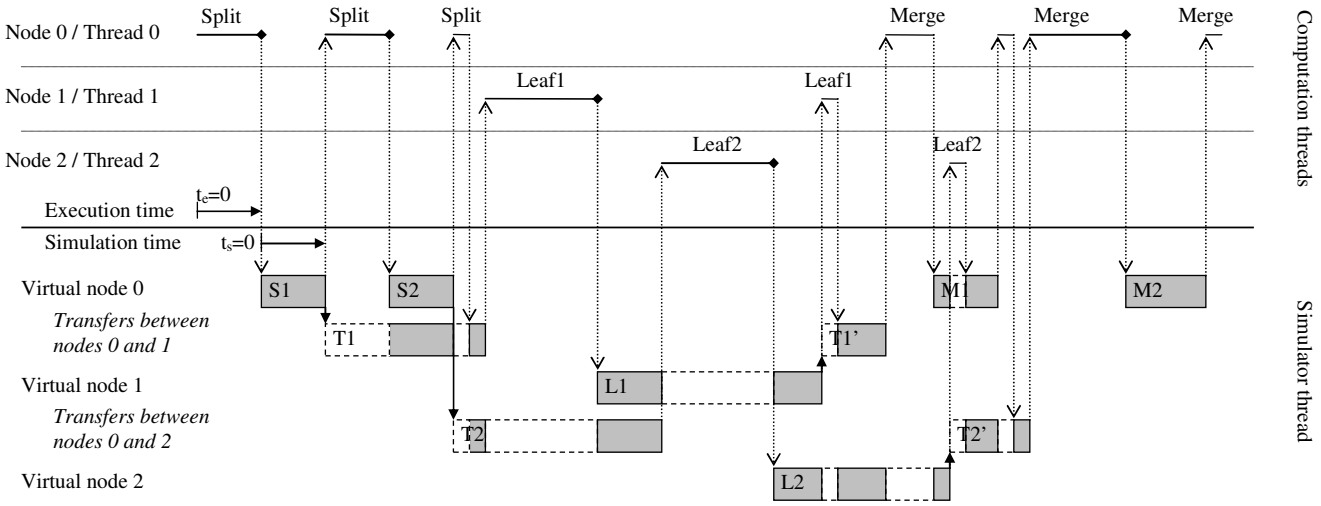

Figure 5. Timing diagram of the simulation of the flow graph shown in Figure 4. The upper part displays the execution of the atomic steps that compose DPS operations. The atomic steps are executed one by one, only when the simulator thread is suspended. The lower part shows the management of the simulated time. Removing the dashed gaps between the gray blocks reveals the timing diagram of Figure 4.

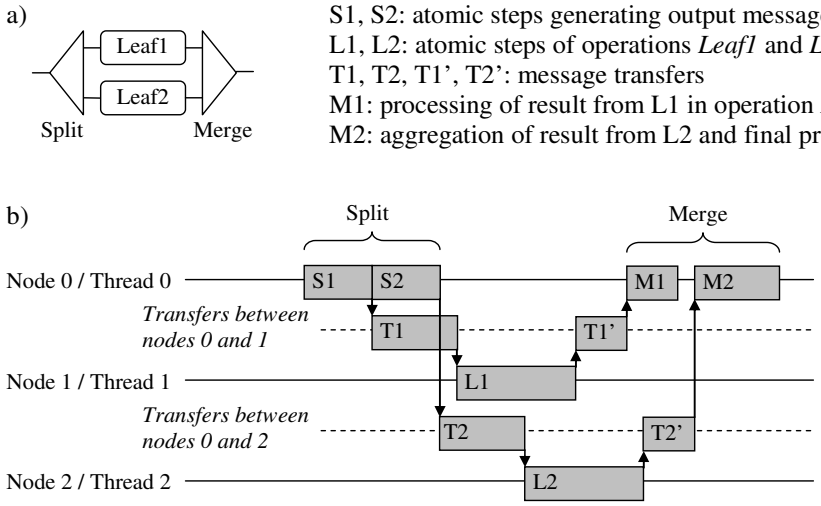

Figure 4. Timing diagram for the parallel execution of a flow graph with a split operation sending two messages to two distinct threads. Each block represents an atomic step. The threads are deployed according to Figure 2.

Figure 5 details the temporal execution of the simulation for the flow graph shown in Figure 4. The simulator thread first triggers the execution of the split operation on *Thread 0*, which runs until the first message is posted and the atomic step S1 and its running time are queued in the simulator. Control is passed to the simulator thread, which increments its simulation clock until the simulation time associated with S1 has elapsed. Then, *Thread 0* is resumed. It first queues the message transfer T1 in the simulator, and resumes execution of the split operation until the second message is sent and the atomic step S2 is queued in the simulator. Although T1 was queued before S2, both atomic steps run in parallel in respect to their simulation time. When S2 completes, control is transferred to *Thread 0* which resumes the split operation. Since no other message must be sent, the split operation terminates, and control returns to the simulator thread. When, within the simulator, the recorded time associated with the message transfer T1 elapses, the associated message is delivered to *Thread 1*, which is resumed and triggers the leaf operation *Leaf1*. The simulation lasts until the final output message of the flow graph is generated.

The upper part of the timing diagram in Figure 5 shows that two computation threads never run simultaneously. The execution of the simulator thread also never overlaps with the execution of the computation threads. In respect to simulation time, operations are correctly overlapping: the timing diagram drawn by the execution of the simulator thread (i.e. with the dashed parts removed) is identical to the timing diagram shown in Figure 4. This simulation scheme also requires no a priori knowledge about the execution: the number of messages, their destination thread, and the number and location of operations are all determined at runtime.

4. The simulator's system model and its assumptions

In the previous section, we have shown that given the running time of each atomic step the parallel structure of the application can be recreated within the simulator. Since only a single computation thread is active at any given time, the processing time of each atomic step can be recorded through direct execution, and be used as its minimal duration, i.e. the running time when CPU or network resources are not shared.

For programs whose parallel execution pattern does not depend on the content of the computed data, the prohibitive running time of direct execution simulation may be reduced by using an estimate of the computation time instead of performing the actual computations. We refer to this technique as *partial direct execution*. The time estimate passed to the simulator is simply a number of microseconds, and may thus come from any source, e.g. deduced from previous executions, computed as a function of some data decomposition parameters, or generated using any other model (see the related work in section 1). By not measuring directly operation execution times, the simulation may run on a computer that is different and potentially less powerful than the one used for the parallel computations.

It is also possible to combine direct execution and partial direct execution. For parallel programs that perform the same operations repeatedly, we may for instance measure the running times of the first n instances of an operation, and reuse the averaged measure for the remaining instances.

The minimal duration of message transfers is estimated using the traditional formula

$$t = l + \frac{s}{b}, \tag{1}$$

where l is the network latency, b the network bandwidth, and s the size of the transferred message. Although the formula is simple, it is very accurate in predicting the TCP/IP transfer time of messages between two processing nodes and has therefore been widely used [3], [14]. It however assumes that no network contention occurs, and can therefore underestimate communication costs for network intensive applications. The latency and bandwidth parameters are constant for a given parallel machine, and must be measured or estimated separately for each target cluster. The size of each message is determined by the simulator at runtime using their size descriptor. The actual message content does not have to be allocated. In partial direct execution simulations, one may therefore avoid allocating the corresponding data structures to reduce the memory requirements of the simulation (the running time of time consuming memory operations can be explicitly added if necessary).

We model resource sharing as follows. We assume that the communication network between the nodes has a star topology, where each node is connected via a full duplex link to a central full crossbar switch

which is never a bottleneck. The input and output bandwidth are both identical and equal to *b*. The bandwidth of each node is shared equally among all incoming, respectively outgoing data transfers. A similar model (with arbitrary topologies) was used in [8]. Transfers between operations running on the same thread or on threads running on the same node are considered to be instantaneous.

Since computations and communications may overlap, the processing power used to handle communications also needs to be taken into account. Receiving messages induces more hardware interrupts and more memory copies than sending messages, and is thus more costly. Moreover, we noticed that the consumed processing power depends on the number of outgoing and incoming communications. Similarly to the bandwidth and latency parameters, the processing power required for communications must be measured separately and provided to the simulator. In all cases, the characterization of these communication and processing parameters is independent of the simulated applications, and thus needs to be carried out only once.

We assume that all nodes have a single processor and that no swapping occurs between memory and disk. Since the simulator has a complete knowledge about ongoing computations and communications, it knows at every time point how many concurrent transfers are carried out by each processing node. It can therefore compute the remaining processing power and distribute it evenly among concurrently running operations. The simulator also produces detailed statistics about the CPU and network usage of each node during application execution.

5. First test application: LU factorization

We first measure the accuracy of our simulator for a parallel block LU matrix factorization application with partial pivoting [11]. The block-based LU factorization relies on the iterative decomposition of the matrix. More or less pipelined implementations improve or degrade the interleaving of operations belonging to successive iteration steps. Such modifications only influence the ordering of the computations, and have no impact on the total amount of data transferred over the network, on the location of the operations, or on the amount of computation they perform. The amount of parallelism and the decomposition granularity of the problem can also be varied, so as to produce executions with different communication pat-

terns and with different computation to communication ratios. Since the amount of computations decreases with every iteration, the efficiency of the application varies over time and can benefit from a reduction in the number of allocated compute nodes. The application therefore provides a wide range of runtime behaviors.

Efficient implementations of the parallel LU factorization use a block-cyclic distribution [5] rather than the parallelization strategy described below. Nevertheless, the higher network utilization of our implementation makes it a good candidate for validating our resource sharing assumptions.

5.1 Implementation

Consider a matrix A of size $n \times n$, with block size r, that is to be factorized. The matrix A is split as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad \text{where } A_{11} \text{ is a square block of size } r$$

$$\begin{matrix} r & n\text{-}r \end{matrix}$$

This matrix is decomposed as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & X \end{bmatrix} \cdot \begin{bmatrix} U_{11} & T_{12} \\ 0 & Y \end{bmatrix}$$

According to this decomposition, the LU factorization can be realized in three steps.

Step 1. Compute the rectangular LU factorization with partial pivoting.

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \cdot U_{11}$$

where L_{11} and U_{11} are lower, resp. upper triangular matrices.

Step 2. Compute T_{12} by solving the triangular system.

$$A_{12} = L_{11} \cdot T_{12}$$

This is the operation performed by the *trsm* routine in BLAS [15]. Carry out row flipping according to the partial pivoting of step 1.

Step 3. To obtain the LU factorization of the matrix A, X must be lower triangular and Y upper triangular. We can define $A' = X \cdot Y$, and iteratively apply the block LU factorization to A' until A' is a square matrix of size r.

$$B = L_{21} \cdot T_{12} + X \cdot Y$$
$$A' = X \cdot Y = B - L_{21} \cdot T_{12}$$

In our implementation, we distribute the matrix onto a set of threads. Each thread stores one column block of size $r \times n$. Another set of threads is dedicated to performing the multiplications of matrix blocks. The flow graph for the LU decomposition is shown in Figure 6. Operation (a) performs the LU factorization of the top left block A_{11} (step 1), and (b) solves in parallel the triangular system in order to compute T_{12} for all other column blocks and performs the row flipping (step 2). The recursion on the matrix factorization is obtained by replicating a part of the graph (in gray) once for each LU factorization level. For the LU factorization presented here, the most expensive part is the block-based matrix multiplication $L_{21} \cdot T_{12}$, both from the computation and the communication perspectives. The multiplication is performed using blocks of size $r \times r$. All input blocks for the multiplication are initially collected within the stream operation (c). The blocks from L_{21} are available on the local thread within which the merge operation is executing, and the blocks from T_{12} are transferred from the local thread states where the preceding *trsm* operations (b) were carried out. The messages sent to each of the matrix block multiplications (d) contain two matrix blocks of size $r \times r$. Messages are routed such that multiplications are evenly distributed on all threads. Each matrix block multiplication yields a matrix block of size $r \times r$ that is sent to the next subtraction operation (e). Notifications are collected at the end of the multiplications (step 3), and as soon as the first block is complete, the next level LU factorization is performed (f). Triangular system solve requests are streamed out as other column blocks complete. Operation (g) performs the row flipping on previous column blocks and the merge operation (h) collects row exchange notifications for termination.

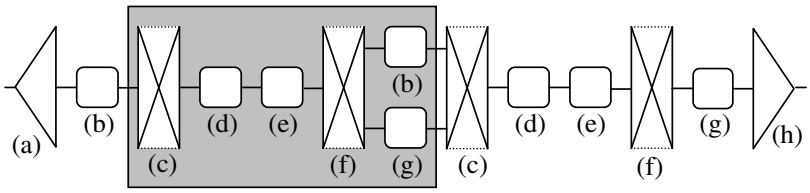

Figure 6. Flow graph for the LU factorization. The gray part is repeated for all but the last iteration.

5.2 Variants

We now explore variations of the decomposition block size, modifications of the LU factorization flow graph and the use of the flow control mechanism provided by the DPS parallelization framework.

In the flow graph of Figure 6, the stream operations (c) and (f) increase the pipelining of the application, i.e. the number of operations that may run concurrently, by allowing *trsm* and *LU* operations (b) and (f) to be performed simultaneously with matrix multiplications (d) and their associated data transfers. We introduce barrier synchronizations by replacing stream operations with merge-split pairs of operations, thereby preventing pipelining. We refer to this less efficient implementation as the *basic* flow graph, as opposed to the *pipelined* flow graph described in Figure 6.

Each thread has an associated queue that stores incoming messages until they are processed. Sending all multiplication requests at once thus fills the queues of the destination threads, which delays the processing of requests sent by subsequent iterations and reduces the pipelining potential. By applying flow control to the stream operations that generate the multiplication requests, we limit the number of messages queued at each iteration. This improves the pipelining by interleaving operations belonging to successive iterations (Figure 7).

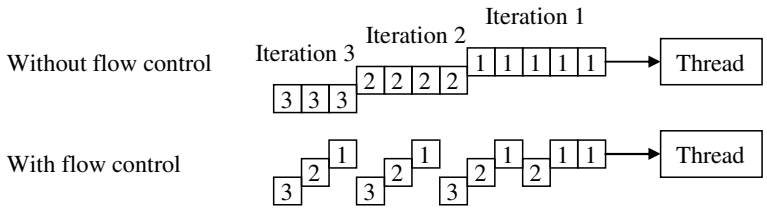

Figure 7. The flow control mechanism improves the interleaving of messages and enables iterations 2 and 3 to be started earlier.

Varying the block size *r* used for the decomposition has an impact on the number of operations, and consequently on the computation to communication ratio (smaller blocks yield a lower computation to communication ratio). In the pipelined flow graph, the value of *r* also influences the depth of the pipeline, and thus the amount of overlapping that can be achieved.

Another modification on the LU factorization flow graph consists in further parallelizing matrix block multiplications by decomposing blocks of size *r* x *r* into row blocks of size *s* x *r* and column blocks of size *r* x *s*. We use a flow graph (Figure 8) that (a) distributes the column blocks of the second matrix to the processing nodes, which (b) store them locally. Each sub-block multiplication can then be performed by (d) sending the line blocks of the first matrix to the processing nodes, which (e) multiply them with the locally stored column blocks. The compositional nature of DPS allows us to replace operation (e) in Figure 6 by the flow graph shown in Figure 8.

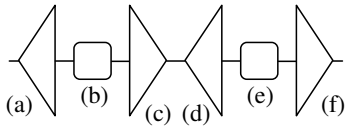

Figure 8. Flow graph for matrix multiplication. It may replace operation (e) in Figure 6.

5.3 Validation

We validate the simulator by comparing measurements and simulations using the parallelization and pipelining flow graph variations discussed in section 5.2. By combining one or several of the modifications proposed and observing their impact on the parallel application's running time, we verify how pre-

the improvement brought by the overlap of communications and computations under various conditions, we also reduced the CPU utilization for the communications. Our first simulations consider eight compute nodes and a coarse decomposition with one column block per node (r=324). The results are summarized in Table 2. Such a decomposition produces fairly large messages and the latency parameter contributes little to their transfer time (line 2), while the bandwidth parameter plays a more important role in the total application running time (line 3). Due to the better overlapping of computations and communications provided by the pipelined flow graph, communication times are partly hidden. Therefore the performance increase brought by the improved network parameters is lower than for the basic flow graph. The factorization of the blocks on the matrix diagonal (operation (f) in Figure 6) lies on the critical path of the execution for the basic flow graph. Speeding up the LU computations by 10% reduces the overall running time by the same duration (4.1 seconds) for both parallelization strategies (Table 2, last line).

	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
Original parameters (r=324)	**86.5**		**78.3**	
Latency=2μs	86.2	-0.3 %	78.1	-0.3 %
Latency=2μs Bandwidth = 912 MB/s	72.7	-16.0 %	69.5	-11.2 %
CPU utilization for comm. divided by 4	82.9	-4.2 %	75.9	-3.1%
LU computation 10% faster	82.4	-4.8 %	74.2	-5.2%

Table 2. Predicted running times with one column block per node on eight nodes (r=324), for varying application and cluster parameters. The relative difference with respect to the predicted running time with the original parameters (in bold) is displayed next to every prediction. The original network is Fast Ethernet, with a latency of 1350μs and a bandwidth of 11.85MB/s.

Table 3 shows the same set of measurements performed when the application runs with a finer grain decomposition (3 column blocks per node, r=108). The total amount of data transferred over the network grows by a factor of 3 (1.3 vs 0.4 GB), and the number of messages increases about 24 times (14701 vs.

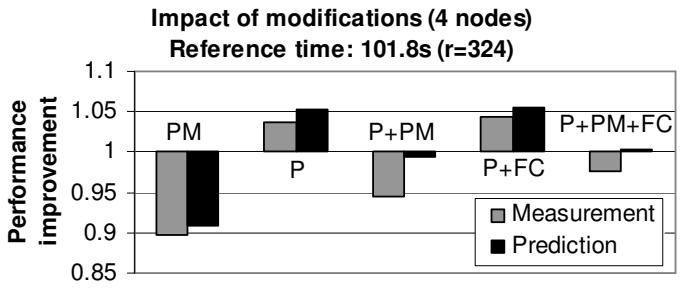

Figure 10. Variation of computation time caused by parallel sub-block multiplications (PM), increased pipelining (P) and flow control (FC), when the matrix is split into two column blocks per node (4 nodes). Prediction errors are below 5%.

Figure 10 shows the effects of the parallel sub-block multiplications (PM), pipelining (P) and flow control (FC) modifications when the matrix is split into eight block columns (i.e. two per node) instead of four, and the reference time is the measured running time when r=324 in Figure 9. Due to the well balanced distribution of block multiplications within the reference setup, the increased communication requirements of transmitting sub-blocks for the parallel sub-block multiplications (PM) slows down the application execution. On the other hand, pipelining (P) and flow control (FC) slightly improve the performance.

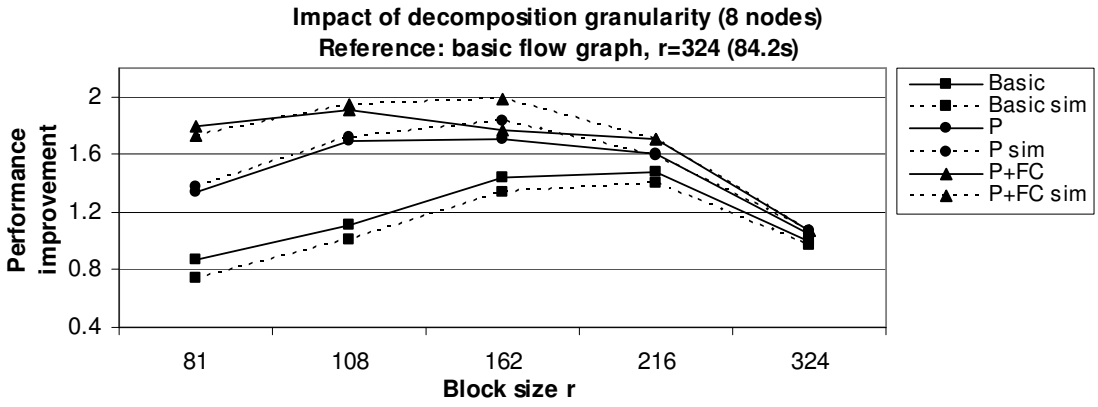

Figure 11. Impact of the decomposition granularity on the performance of different pipelining strategies (8 nodes).

When we increase the number of processing nodes to eight nodes, the benefits of the pipelined flow graph (P) and of the flow control (FC) become more significant (Figure 11). The optimal block size for the LU factorization is also influenced by the parallelization strategy. In all cases, pipelining considerably improves the performance with respect to the basic flow graph, and the conjunction of pipelining and flow control further improves the results. Note that the growth in the number of operations performed and messages sent during execution (from 352 when r=324 to about 22,000 when r=81) has no visible impact on the prediction accuracy.

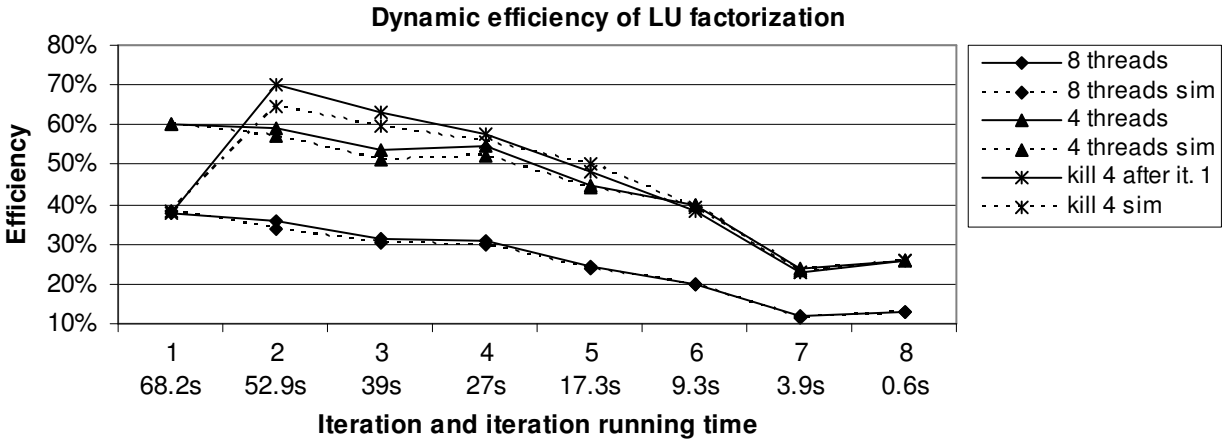

Figure 12. The parallel computation of LU iterations becomes less efficient over time. Removing threads during execution increases the efficiency of the subsequent iterations.

We now consider the impact of reducing the number of multiplication threads during execution. In our test case, the 2592x2592 matrix is split into eight column blocks distributed onto four nodes (r=324), and the computation is performed using the basic flow graph, allowing to clearly separate the different iterations. Figure 12 shows the dynamic efficiency (i.e. the efficiency at each iteration step) of the application. During the first iteration, four nodes are about 50% more efficient than eight nodes (60.2% vs. 37.6%). The relative efficiency of 4 nodes versus 8 nodes increases up to iteration 6 where 4 nodes have twice the efficiency of 8 nodes, i.e. iteration 6 has the same running time on 4 nodes and on 8 nodes. Therefore, removing nodes during execution should not have a large impact on the total computation time.

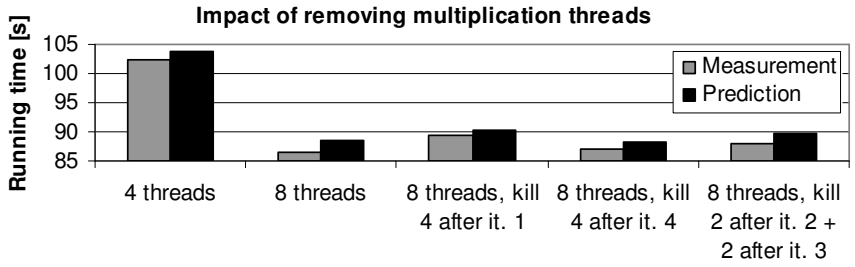

Figure 13. Measured and predicted running times of different dynamic thread removal strategies.

This is confirmed by measuring the total execution time of the application for different thread removal strategies (Figure 13). Using eight nodes for the whole computation or only for the first iteration yields almost the same running time, and being able to deallocate four nodes after the first iteration greatly increases the dynamic efficiency of the application (Figure 12, "kill 4 after iteration 1"). Figure 14 displays the real and simulated trace of the corresponding computations (network transfers are hidden for readability).

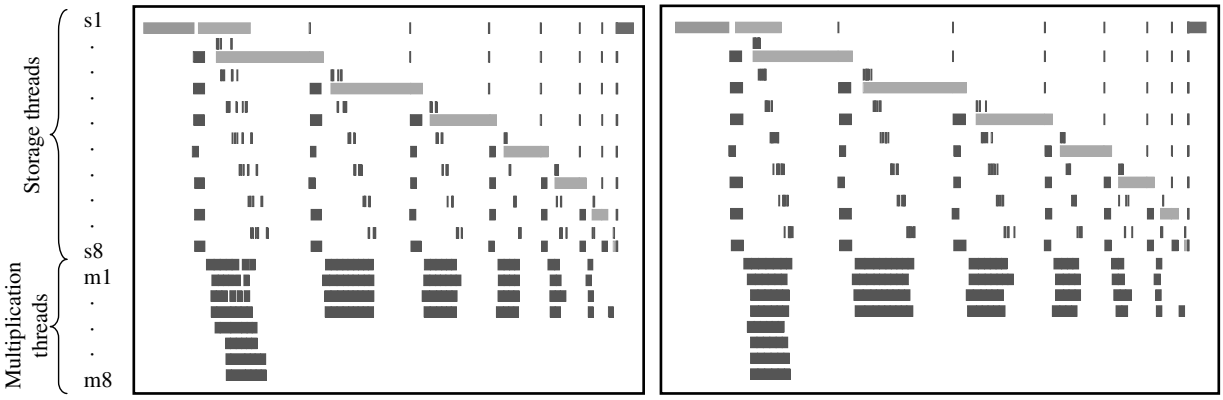

Figure 14. Trace of the real (left) and simulated (right) execution of the "kill 4 after iteration 1" configuration in Figure 12 (network transfers are not shown). Time runs from left to right. The first eight pairs of lines represent operations running on threads s1-s8 that store column blocks. The last eight lines represent operations on multiplication threads m1-m8 (dark gray), four of which are removed after the first iteration. All the other threads run on the four remaining compute nodes.

Since the first iteration accounts for approximately 25% of the parallel running time, the service rate of the cluster can be significantly increased if the deallocated compute nodes are assigned to other applications. In this example, the execution with the static node allocation uses eight nodes during 86.9 seconds, or 695.2 seconds, while the dynamic allocation strategy requires eight nodes during 22.5 seconds and four nodes during 66 seconds. The total processor utilization is therefore reduced by 37% to 438.4 seconds.

This is confirmed by measuring the total execution time of the application for different thread removal strategies (Figure 13). Using eight nodes for the whole computation or only for the first iteration yields almost the same running time, and being able to deallocate four nodes after the first iteration greatly increases the dynamic efficiency of the application (Figure 12, "kill 4 after iteration 1"). Figure 14 displays the real and simulated trace of the corresponding computations (network transfers are hidden for readability).

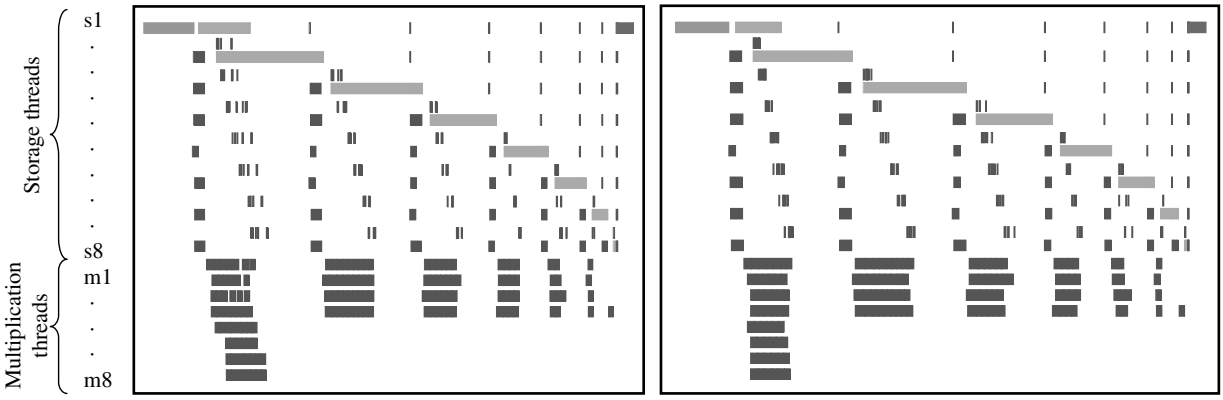

Figure 14. Trace of the real (left) and simulated (right) execution of the "kill 4 after iteration 1" configuration in Figure 12 (network transfers are not shown). Time runs from left to right. The first eight pairs of lines represent operations running on threads s1-s8 that store column blocks. The last eight lines represent operations on multiplication threads m1-m8 (dark gray), four of which are removed after the first iteration. All the other threads run on the four remaining compute nodes.

Since the first iteration accounts for approximately 25% of the parallel running time, the service rate of the cluster can be significantly increased if the deallocated compute nodes are assigned to other applications. In this example, the execution with the static node allocation uses eight nodes during 86.9 seconds, or 695.2 seconds, while the dynamic allocation strategy requires eight nodes during 22.5 seconds and four nodes during 66 seconds. The total processor utilization is therefore reduced by 37% to 438.4 seconds.

favorable. This leads to a lower speedup, despite the larger running time of the application. The speedup predicted by simulation and the actually measured speedup differ by 5.3% on average.

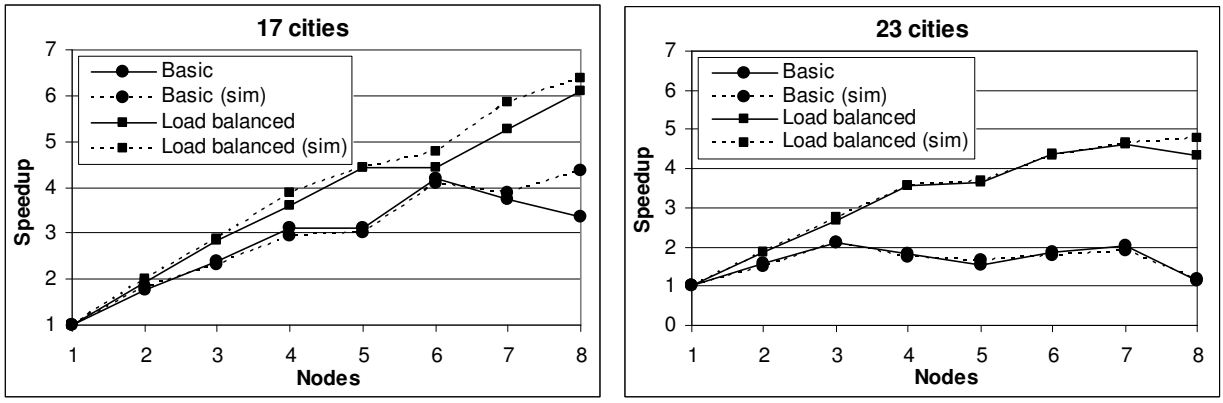

Figure 15. Measured and predicted speedups for a traveling salesman solver.

7. Improving simulation times and portability through partial direct execution

In the present section, we analyze to which extent the simulation time and memory use can be reduced by partial direct execution.

Table 1 displays the time required to perform the simulation of the LU factorization of a 2592x2592 matrix, with the real application running on eight nodes, using the basic flow graph and the decomposition granularity r=216. For reference, the real parallel execution lasts 62.3s, and the real serial execution lasts 185.1s. With a running time of 193s, the simulator's overhead when direct execution is used is 4.3%.

We implement partial direct execution (PDEXEC) by simply replacing calls to the matrix multiplication, LU, *trsm*, and row flipping functions with simulator notifications incorporating the corresponding benchmarked times. We then remove the memory allocation for the initial matrix (NOALLOC), together with memory copies performed in the corresponding DPS operations. The final simulation is almost ten times faster than the actual parallel execution on the same hardware and uses only 14MB of memory. The predicted running time changes by only -1.3% compared with the direct execution simulation.

UltraSparc II 440Mhz (Solaris)	Running time [s]	Memory usage [MB]	Predicted running time [s]
Real application (8 nodes)	**62.3**		N/A
Real application (1 node)	185.1	108	N/A
Direct execution (sim)	193.0	127	60.7
PDEXEC (sim)	9.1	124	60.3
PDEXEC NOALLOC (sim)	6.5	14	59.9
Pentium 4 2.4GHz (Windows)			
Direct execution (sim)	29.7	127	N/A
PDEXEC (sim)	2.5	124	60.0
PDEXEC NOALLOC (sim)	1.6	14	59.9

Table 1. Comparison of simulation times and memory consumptions in different simulation settings, and corresponding predicted running time. The real application running time is 62.3 s (in bold).

This optimized simulator mode produced all the simulation results presented in section 5.3. Its prediction accuracy for the 168 measurements carried out for establishing the results are shown in Figure 16. 71.4% of all predictions are within ±4% accuracy, 81.6% are within ±6% accuracy, and more than 95% are within ±12% prediction accuracy.

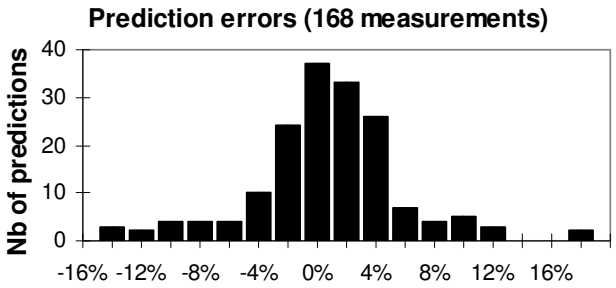

Figure 16. Histogram of prediction errors with partial direct execution

Table 1 displays simulation results for two different platforms. Since the Pentium 4 processor is much faster than the UltraSparc II, prediction results based on direct execution are not representative. However, when partial direct execution is used, the faster processor has nearly no impact on the predicted running time of the LU factorization application. In order to assess the portability of our simulator, we ran a same set of simulations on four different systems, three with single processors at 600MHz, 2.4GHz and 3GHz, and on one with two dual-core 2.6GHz processors. The simulation set consists of 100 different application configurations, combining different number of nodes, decomposition block sizes, the use of flow control

and parallel sub-block multiplications. We ran all simulations with and without matrix allocation (NOALLOC), producing 200 prediction results. Figure 17 shows the relative difference of the 200 predictions produced by each one of the three fastest systems, compared to the predictions produced by the slowest system (600 comparisons in total). Despite the performance difference, 97% of the prediction results differ by only ±2%. The outliers with an error greater than 5% represent 1.3% of all measurements. The fact that predictions made on the multiprocessor system match results obtained on single processor systems shows that the execution of the various computation threads is correctly sequenced.

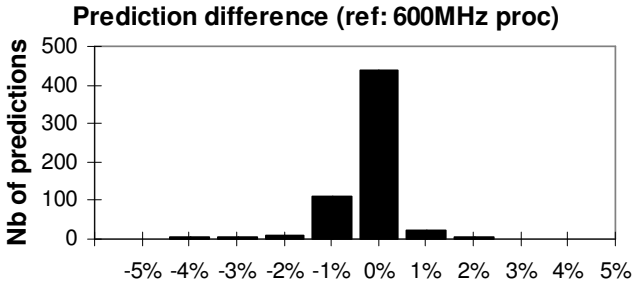

Figure 17. Histogram of relative differences of running times predicted on the three fastest systems with respect to predictions performed on the slowest system (3 times 200 comparisons in total).

8. Using the simulator for analyzing parallel applications

As described in the previous section, modeling the duration of the individual operations and message transfers of a DPS application decreases the running time and memory consumption of the simulated application. It also leads to a parametric model of the application [14]. Since parametric models allow the different performance factors to be isolated from one another, they enable analyzing the sensitivity of the overall running time with respect to the different parameters. Varying the running time of specific operations helps identifying the operations located on the critical path of the computation and quantifying the potential benefits of their optimization.

For both the basic and the pipelined flow-controlled LU factorization application, we simulated a high performance network by reducing the latency and increasing the bandwidth parameters. In order to study

the improvement brought by the overlap of communications and computations under various conditions, we also reduced the CPU utilization for the communications. Our first simulations consider eight compute nodes and a coarse decomposition with one column block per node (r=324). The results are summarized in Table 2. Such a decomposition produces fairly large messages and the latency parameter contributes little to their transfer time (line 2), while the bandwidth parameter plays a more important role in the total application running time (line 3). Due to the better overlapping of computations and communications provided by the pipelined flow graph, communication times are partly hidden. Therefore the performance increase brought by the improved network parameters is lower than for the basic flow graph. The factorization of the blocks on the matrix diagonal (operation (f) in Figure 6) lies on the critical path of the execution for the basic flow graph. Speeding up the LU computations by 10% reduces the overall running time by the same duration (4.1 seconds) for both parallelization strategies (Table 2, last line).

	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
Original parameters (r=324)	**86.5**		**78.3**	
Latency=2μs	86.2	-0.3 %	78.1	-0.3 %
Latency=2μs Bandwidth = 912 MB/s	72.7	-16.0 %	69.5	-11.2 %
CPU utilization for comm. divided by 4	82.9	-4.2 %	75.9	-3.1%
LU computation 10% faster	82.4	-4.8 %	74.2	-5.2%

Table 2. Predicted running times with one column block per node on eight nodes (r=324), for varying application and cluster parameters. The relative difference with respect to the predicted running time with the original parameters (in bold) is displayed next to every prediction. The original network is Fast Ethernet, with a latency of 1350μs and a bandwidth of 11.85MB/s.

Table 3 shows the same set of measurements performed when the application runs with a finer grain decomposition (3 column blocks per node, r=108). The total amount of data transferred over the network grows by a factor of 3 (1.3 vs 0.4 GB), and the number of messages increases about 24 times (14701 vs.

New Text

613). The lower computation to communication ratio induced by smaller blocks causes network transfers to account for a greater part of the overall running time. Since messages are smaller, the network latency also becomes an important factor. Both considerations are reflected in the simulation results, where improved network parameters reduce the running time much more than with the coarser decomposition used in Table 2. Their impact is smaller, but remains important, for the pipelined flow graph.

On the hardware used for our real execution measurements, handling multiple simultaneous transfers to eight nodes requires more than 50% CPU utilization. This factor is very important for the pipelined flow graph due to the large overlap between communications and computations. Dividing this CPU utilization for communications by four therefore significantly decreases the application running time (-27% in Table 3, line 4). The increased decomposition granularity reduces the weight of the LU factorization operations. Since the network is now the bottleneck, optimizing these computations yields very little benefits for both flow graphs (Table 3, last line).

	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
Original parameters (r=108)	**83.9**		**43.0**	
Latency=2μs	77.2	-8.0 %	41.2	-4.1 %
Latency=2μs Bandwidth = 912 MB/s	30.4	-63.7 %	24.9	-42.0 %
CPU utilization for comm. divided by 4	83.1	-0.9 %	31.2	-27.4 %
LU computation 10% faster	83.2	-0.8 %	42.6	-1.0 %

Table 3. Predicted running times with three column blocks per node on eight nodes (r=108), for varying application and cluster parameters. The relative difference with respect to the predicted running time with the original parameters (in bold) is displayed next to every prediction.

The impact of the excessive network utilization of our parallel LU factorization implementation is even more apparent when we simulate faster processors by dividing all computations times by four and by reducing the CPU consumed by communications by a factor of four. Table 4 shows results for both r=324 and r=108. Improving the latency and the network bandwidth now yields very significant running times

reductions in all configurations. As expected, the faster processors reduce running times in all cases. However, the basic flow graph now runs faster with the coarser and less network-intensive decomposition (one column block per node, r=324). As expected, the pipelined flow graph performs better than the basic flow graph.

	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Difference in respect to original parameters
r=324, 4x faster processors	**36.6**		**25.4**	
Latency=2μs	36.3	-0.9 %	25.3	-0.3 %
Latency=2μs Bandwidth = 912 MB/s	18.3	-49.9 %	17.5	-31.0 %
r=108, 4x faster processors	**76.2**		**24.2**	
Latency=2μs	69.3	-9.0 %	21.0	-13.2 %
Latency=2μs Bandwidth = 912 MB/s	8.3	-89.1 %	6.6	-72.5 %

Table 4. Impact of network parameters on predicted running times when the duration of all individual computations and the CPU consumption of communications have been reduced by a factor of 4.

The presented results show that each one of the selected hardware parameters, i.e. the network latency, network bandwidth and the CPU consumption for communications, has a significant impact on the application running time. The quality of the predictions obtained in the previous sections show that this parameter set is sufficient for characterizing the behavior of a cluster composed of a small set of computing nodes.

Despite the approximations made within the models and within the simulations, our simulator can be used as a performance analysis tool. Several tiny delays, such as the internal latencies of the parallel runtime system, are neglected in the current model. The accuracy of predictions is therefore likely to decrease for fine-grain applications performing many very short operations and sending many very small messages.

9. Conclusions and future work

The performance of a parallel application not only depends on its implementation, but also on decomposition parameters, on tasks to nodes mapping and on node allocation decisions. The choice of optimal

parameters may depend on the application input data as well as on the number of allocated compute nodes. The dynamic allocation of compute nodes during the execution of parallel applications can further improve the utilization of cluster resources. In order to help decide how and when the allocation should be modified, we introduce the concept of dynamic efficiency which expresses the resource utilization efficiency as a function of time. We obtain information about the performance and the dynamic efficiency of parallel programs by running a simulator on top of the parallelization framework runtime system.

In the presently used Dynamic Parallel Schedules framework, the parallel structure of an application is specified by a flow graph where nodes represent serial computations and edges represent transferred messages. Computations are performed by threads, which can be dynamically allocated or deallocated onto compute nodes. We simulate the parallel execution of an application by running all threads within a single application instance. The simulator then coordinates and synchronizes the execution of the threads to control the application execution. Communication patterns, as well as the number of messages and operations are derived through direct execution.

By default, the duration of each operation is also obtained through direct execution. The running time, memory requirements and portability of the simulation are improved by using partial direct execution, i.e. by replacing time-consuming computations with running time predictions, and by avoiding large memory allocations. Varying the duration of individual operations enables determining the operations that belong to the critical path and that can benefit from further optimizations.

We describe a simple model for typical cluster configurations that accurately takes bounded and shared network and CPU resources into account. We verify the prediction accuracy of our simulator by applying several parallelization and deployment strategies to an LU factorization application and to a simple traveling salesman problem solver. The LU factorization application also shows that the simulator is able to accurately predict running times and dynamic efficiency when deallocating compute nodes at different time points of the program execution. By varying the simulated hardware parameters such as the processing power of the compute nodes, the network latency and throughput, and the CPU utilization of network

communications, we identify the performance bottlenecks within the application and validate our cluster parameterization and resource sharing model.

Although results are presented here in the context of DPS, the cluster modelization and the principles of the simulator can be adapted to other parallelization models.

In the future, we intend to extend the simulation framework in order to simulate a cluster running concurrently multiple, possibly different applications whose compute nodes allocation varies dynamically over time.

10. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] V. S. Adve, M. K. Vernon, *Parallel program performance prediction using deterministic task graph analysis*, ACM Transactions on Computer Systems (TOCS), Vol. 22 , No. 1, pp. 94-136, February 2004

[2] V. D. Agrawal, S. T. Chakradhar, *Performance estimation in a massively parallel system*, Proc of Supercomputing '90, pp 306-313, Nov. 1990

[3] C. Anglano, *Predicting parallel applications performance on non-dedicated cluster platforms*, Proc. 12th Int'l Conference on Supercomputing, Melbourne, Australia, pp. 172-179, 1998

[4] R. Bagrodia, E. Deeljman, S. Docy, T. Phan, *Performance prediction of large parallel applications using parallel simulations*, Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pp. 151-162 , 1999

[5] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, R. C. Whaley, *The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines*, Scientific Programming, Vol. 5, pp. 173-184, 1996

[6] W. Cirne, F. Berman, *A model for moldable supercomputer jobs*, Proc. 15th Int'l Parallel and Distributed Processing Symposium, April 2001

[7] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, S. Madala, *The efficient simulation of parallel computer systems*, International Journal in Computer Simulation, Vol. 1, 1991

[8] B. Hong, V.K. Prasanna, *Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput*,, Proc. 18th Int'l Parallel and Distributed Processing Symposium, p. 52b, 2004.

[9] S. Gerlach, R. D. Hersch, *DPS - Dynamic Parallel Schedules*, Proc. 8th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003), 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, pp. 15-24, April 2003

[10] S. Gerlach, R.D. Hersch, *Fault-tolerant Parallel Applications with Dynamic Parallel Schedules*, Proc. 19th International Parallel and Distributed Processing Symposium (IPDPS), 10th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'05), April 4-8, Denver CO, USA, p. 278b, 2005

[11] G. H. Golub, C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, pp. 94-116, 1996

[12] R. Gruber, V. Keller, P. Kuonen, M.-Ch. Sawley, B. Schaeli, A. Tolou, M. Torruella, T.-M. Tran, *Intelligent GRID Scheduling System*, Proc. 6th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM'05), Poznan, Poland, Sept. 2005

[13] L. V. Kale, S. Kumar, J. DeSouza, *A malleable-job system for timeshared parallel machines*, 2nd IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGRID'02), pp. 215-222, May 2002

[14] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, M. Gittings, *Predictive performance and scalability modeling of a large-scale application*, Conference on High Performance Networking and Computing, Proc. 2001 ACM/IEEE Conference on Supercomputing, Denver, Colorado, pp. 37-37, 2001

[15] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft., Vol. 5, pp. 308-323, 1979

[16] D.-R. Liang, S. K. Tripathi, *On performance prediction of parallel computations with precedent constraints*, IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 5, pp. 491-508, May 2000

[17] G. D. Peterson, R. D. Chamberlain, *Stealing cycles: Can we get along?*, Proc. 28th Hawaii Int'l Conf. on System Sciences, Vol.2, pp. 422-431, Jan. 1995

[18] S. Prakash, R. Bagrodia, *MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs*, Proc. 1998 Winter Simulation Conference, 1998

[19] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, A. Purkayastha, *A framework for performance modeling and prediction*, Proc. 2002 ACM/IEEE conference on Supercomputing, pp. 1-17, Baltimore, Maryland, 2002

[20] G. Utrera, J. Corbalan, J. Labarta, *Implementing malleability on MPI jobs*, Proc. 13th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT'04), pp. 215-224, Oct. 2004