A simulator for parallel applications with dynamically varying compute node allocation

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. We introduce the concept of dynamic efficiency which expresses the resource utilization efficiency as a function of time. We propose a simulation framework which enables predicting the dynamic efficiency of a parallel application. It relies on the DPS parallelization framework to which we add direct execution simulation capabilities. The high level flow graph description of DPS applications enables the accurate simulation of parallel applications without needing to modify the application code. Thanks to partial direct execution, simulation times and memory requirements may be reduced. In simulations under partial direct execution, the application's parallel behavior is simulated thanks to direct execution, and the duration of individual operations is obtained from a performance prediction model or from prior measurements. 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, respectively the dynamic efficiency, predicted by the simulator. These comparisons are performed for an LU factorization application under different parallelization and dynamic node allocation strategies.

1. Introduction

Recent studies show that many parallel applications do not fully use the available hardware [6, 9]. Since in most parallel systems a constant number of nodes is allocated to an application, nodes may become idle or underutilized when the application's processing power requirements vary over the course of execution. Therefore, one may increase the utilization of computing resources during program execution by adapting the allocation of computing nodes to the applications' computation needs. For example, the amount of computation performed by an LU factorization application decreases at each iteration of the algorithm. The number of allocated nodes may thus be decreased over the course of execution without significantly increasing the execution time of the parallel application.

The DPS parallelization framework [7] provides the functionality required to modify the allocation of processing nodes to an application at runtime. However, taking the right decisions requires a priori knowledge about the *dynamic efficiency* of the application, i.e. its utilization of resources as a function of time. Detailed simulations of the application can provide means of capturing that information, together with information about the effectiveness of the chosen problem decomposition and allocation of processing nodes.

In order to obtain information on the dynamic efficiency and to simulate the influence of parallel application parameters, we integrated our simulator within the DPS parallelization framework. Simulating a parallel application by executing the application code at least partially allows to reconstruct its exact behavior and to predict its parallel running time, given a representation of the running time of each of its tasks. Since the DPS runtime code is executed during the simulation, its dynamic features such as the dynamic allocation of processing nodes can also be simulated. Therefore, the impact of different parallelization and deployment strategies on the application running time can be evaluated.

Purely analytical models for the performance prediction of parallel programs are generally tailored to a specific application [11] or to a class of parallel programs, such as fork-join applications [14]. 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, 13] or as a timed Petri net [3]. Approaches for the purpose of modeling individual task execution times include measurements [3, 11], stochastic models [13, 14] and the association of an application signature and a machine profile [16]

MPI-SIM [15] and its extension COMPASS [4] are two simulators that predict the performance of MPI programs by executing the application. The simulation functionality is provided by a modified library that implements the most common MPI calls. Both MPI-SIM and COMPASS use direct execution [6] to derive computation times. Direct execution does not require any modification to the application. However, it has the drawbacks that the simulation must run on the same hardware that runs the parallel application and that 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 availability of the parallel machine for the simulation.

We follow a mixed approach, where the task-level behavior and task execution times are derived through the use of *partial direct execution*. Computations that have no impact on the task-level behavior of the application may be replaced by duration estimates. In addition, we may also reduce memory usage by avoiding data structure allocations. The direct execution drawbacks are therefore considerably reduced. Moreover, unlike other simulators which ignore network delays [2, 14] or assume that

20th International Parallel and Distributed Processing Symposium (IPDPS)
5th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems
(PMEO-PDS'06), April 25-29, Rhodes Island, Greece, 2006

network contention is inexistent [4, 15], 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 can accurately simulate the execution of parallel programs on a desktop computer.

The problem of dynamically allocating resources to parallel applications has been previously considered [5, 10, 17]. However, according to our knowledge the simulator we propose is the first one which predicts the performance of real adaptive applications, i.e. applications whose mapping to computation nodes may vary over time during program execution.

2. The Dynamic Parallel Schedules framework

DPS applications are defined as directed acyclic graphs of operations [7]. Its fundamental types of operations are the *leaf*, *split*, *merge* and *stream* operations. The inputs and outputs of the operations are strongly typed data objects. Figure 1 illustrates the flow graph of a simple parallel application. The flow graph describes the asynchronous flow of data between operations.

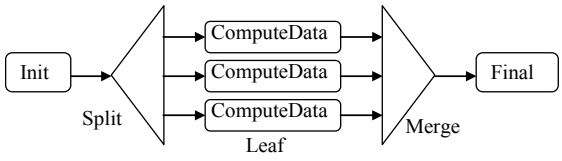

Figure 1. Flow graph describing data distribution (split), parallel processing, and collection of results (merge)

The *split operations* are used to divide the incoming data objects into smaller objects representing subtasks. These subtasks are subsequently sent to the next operations specified by the flow graph (e.g. ComputeData in Figure 1). The *merge operations* are used to collect and aggregate the results into a single output object. Once all the results corresponding to the data objects originally sent by a split operation have been collected, the resulting data object is sent out. Successive data objects arriving at the entry of a split operation yield successive new instances of the split-merge operation pair.

The *stream operations* combine a merge operation with a subsequent split operation. Instead of waiting for the merge operation to receive all its data objects before allowing the subsequent split operation to send new data objects, the stream operation can stream out new data objects based on groups of incoming data objects. By refining the synchronization granularity, stream operations allow programmers to maximize the pipelining of parallel operations, thereby ensuring a maximal utilization of the underlying hardware.

All operations are extensible constructs, i.e. the developer provides his own code to control how processing requests are split into sub-requests, how the data is distributed and processed, and how processed sub-results are merged into one result. The data objects circulating in the flow graph may contain any combination of simple types or complex types such as arrays or lists.

Operations within a flow graph are carried out within *threads.* A thread in DPS is a logical construct representing an execution environment for a set of operations. DPS threads are mapped onto operating system threads, called *DPS execution threads*, although not necessarily in a one-to-one relationship. For instance several DPS threads residing on a single processor node may share a single DPS execution thread.

The selection of the DPS thread on which an operation is to be executed is accomplished by evaluating at runtime a user defined routing function attached to the corresponding directed edge of the flow graph. Communication patterns such as neighborhood exchanges can easily be specified by using relative thread indices.

By transferring data objects as soon as they are generated and by maintaining queues of arriving data objects, the execution of DPS applications is fully pipelined and asynchronous. Data object queues are associated with the thread that contains the operations that will consume them. This macro data flow behavior enables automatic overlapping of communications and computations. A flow control mechanism can be used to limit the number of data objects in circulation between a split and the corresponding merge operation. This prevents split and stream operations from filling the data object queue of the destination threads.

The deployment of a DPS application is done at runtime, and relies on a remote launching mechanism to create a new application instance on every node that will host a DPS thread. In each application instance, a *DPS thread manager* handles thread creation and destruction requests, and delivers incoming data objects to their destination thread.

The flow graph together with its threads and its routing functions forms a *parallel schedule*. A parallel schedule describes a fine to medium-grained parallel application. Its operations represent the small subtasks that are executed in a pipeline-parallel manner according to the flow graph. The DPS communication layer, hidden from the application programmer, relies on TCP sockets, and uses an optimized data serialization scheme that minimizes memory copies.

3. Structure of the DPS simulation system

The DPS flow graph only gives a logical description of the parallel behavior of an application. The simulation of a parallel application requires additional run time information to be able to precisely reconstruct the actual execution. The number of processing nodes and threads must be known at every moment, along with the functions that route data objects onto threads, and the number of data objects sent by each split or stream operation.

Since the simulator is integrated within the DPS parallelization framework, it has access to all the parameters that have an impact on the execution of a parallel application by directly executing code from both the application and the DPS runtime.

The simulation of the deployment of DPS threads onto computing nodes is carried out as follows. A modified remote launching mechanism instantiates a new DPS thread manager for each application instance that would have been launched in a real execution. Simultaneously, the simulator maintains a virtual representation of each computing node on which the application is deployed. The TCP network layer is replaced by a simulated network layer, which handles all communications between the virtual nodes. Since the network layer is fully simulated, the mechanisms that create and destroy DPS threads may be used without any modification. Hence, the simulation of an application uses the same number of DPS thread managers and the same deployment scheme as the real execution. The only difference is that all thread

managers are running within the simulator. The simulator is therefore able to reconstruct the actual application execution by keeping track of which thread, and thus which virtual node, executes which operation.

DPS operations may be suspended during their execution, e.g. when merge and stream operations wait for data objects that did not yet arrive, or due to the DPS flow control mechanism. We therefore subdivide operations into *atomic steps*, i.e. operation parts which execute without being suspended. An atomic step starts when another atomic step is completed, and ends when a data object is posted or when an operation is suspended or terminates. Since data transfers cannot be suspended, they are also assimilated to atomic steps.

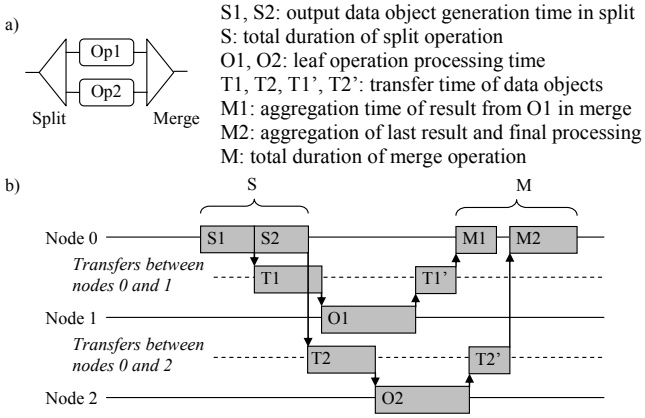

Figure 2. Timing diagram for the parallel execution of a flow graph deployed on three nodes with a split operation sending two data objects. Each block represents an atomic step.

Figure 2 shows the atomic steps of the execution of a simple flow graph on 3 nodes, one node running the split and merge operations, the other two running the leaf operations. The split operation is composed of the atomic steps S1 and S2, which respectively generate the data object transfers T1 and T2. Each leaf operation is made of a single atomic step (O1 and O2). The resulting data object transfers T1' and T2' trigger the execution of the atomic steps M1 and M2 within the merge operation. The gap between M1 and M2 indicates that the merge operation is suspended, waiting for the data object (result) created by the leaf operation O2 on node 2.

The overlap of communications and computations is maximized by running different operations on distinct DPS execution threads, allowing for example a merge operation to receive and process data objects while a leaf operation is running on the same processor. In order to accurately measure execution times during a direct execution simulation, the simulator has to control the activation of execution threads and ensure that only one of them is active at any given moment. This is done by running the simulator code in an operating system thread (called simulator thread) distinct from the DPS execution threads. At points within the DPS framework code that terminate an atomic step, notifications inform the simulator that an atomic step has been carried out and that the corresponding running time needs to be recorded. The running DPS execution thread is then suspended and control is passed to the simulator thread (Figure 3).

Each atomic step is recorded and stored into the simulator with a measurement or an estimate of its duration. When the simulator thread is running, it looks for the recorded atomic step that completes next, and advances accordingly its simulation clock. The DPS execution thread associated to the completed recorded atomic step is resumed, and the simulator thread is suspended. If the completed atomic step represents a data object network transfer, the resumed execution thread is the one that received the transferred data object. If the atomic step represents a computation, the resumed execution thread is the one running the corresponding operation.

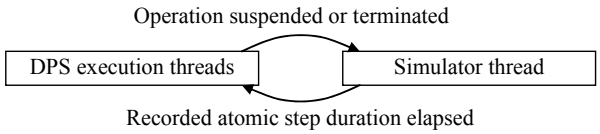

Figure 3. Alternating DPS operation direct execution and simulator execution steps.

After execution of the next atomic step by the DPS execution thread, control is returned to the simulator thread in order to record the atomic step's running time and advance the simulation clock. Therefore, all the atomic steps are executed sequentially and their contribution to the application's running time can be correctly recorded.

Figure 4 shows the temporal execution of the simulation for the flow graph shown in Figure 2. The simulator thread first triggers the execution of the split operation ($split_1$), which runs until the first data object is posted and the running time record of atomic step S1 is queued in the simulator. Control is passed to the simulator thread, which increments its simulation clock until the simulation time corresponding to S1 has elapsed. Then, the DPS execution thread is resumed. It first queues the data object transfer T1 in the simulator, and resumes execution of the split operation ($split_2$) until the second data object is posted 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 the DPS execution thread which immediately terminates the split operation. The DPS execution thread then passes control to the simulator thread, and is suspended waiting for another data object to process. When, within the simulator, the recorded time associated with the data object transfer T1 elapses, the associated data object is delivered to the DPS execution thread running on virtual node 1. Control is passed to the DPS execution thread, which triggers the leaf operation Op1.

The upper part of the timing diagram in Figure 4 shows that two DPS execution threads never run simultaneously. The simulator thread also never overlaps with DPS execution 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 2.

Since the simulation library is integrated into DPS, the simulated application is obtained by simply activating a compilation flag. The real and simulated applications may thus be run identically, and the command line arguments (which may for instance specify the number of nodes to be used or the decomposition

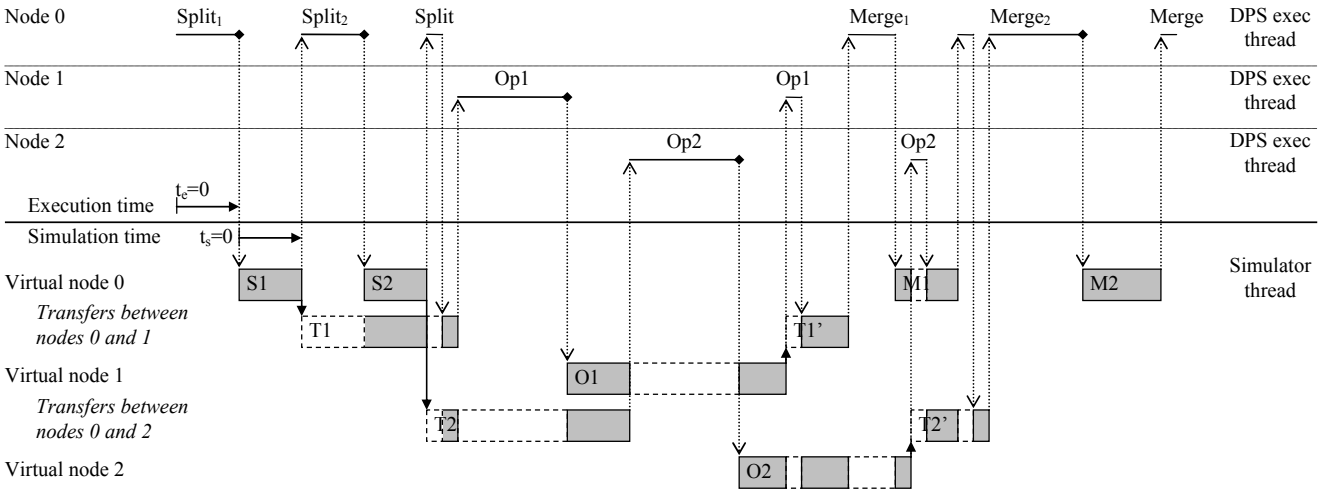

Figure 4. Timing diagram of the simulation of the flow graph shown in Figure 2. The execution of DPS operations is shown in the upper part and the management of the simulated time in the lower part. Atomic steps within DPS operations are executed one by one, only when the simulator thread is suspended.

granularity) will have the same effect on both versions of the program.

4. The simulator's system model and its assumptions

In the previous section, we have shown that the parallel structure of the application can be recreated within the simulator, given the running time of each atomic step. Since only a single operating system 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 optimistic running time, i.e. the running time when assuming that there is no CPU or network contention.

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 passing an estimate of the computation time instead of performing the actual computations. We refer to this technique as partial direct execution. The time estimate is simply a number of microseconds, and may thus come from any source, i.e. either deduced from previous executions, computed as a function of some parameters, or generated using any other model (see the related work in section 1). It is also possible to combine direct execution and partial direct execution. For parallel programs that perform the same operations repeatedly, we may measure the running times of the first n instances of an operation, and reuse the averaged measure for the remaining instances.

By avoiding time measurements during program execution, the hardware running the simulation no longer impacts the predicted running time of the simulated application's operations. The use of partial direct execution therefore enables the simulation to run on a computer that is different and potentially less powerful than the one used for the parallel computations.

The optimistic time for data object transfers are estimated using the traditional formula

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

where l is the network latency, b the network bandwidth, and s the size of the transferred data object. Although the formula is simple, it is very accurate in predicting the TCP/IP transfer time of a data object between two processing nodes and has therefore been widely used [3, 11]. The latency and bandwidth parameters are constant and specific to the hardware onto which the parallel application is running. They must therefore be measured or estimated separately for each target parallel machine. The size of the data objects is computed at runtime, using a modified version of the built-in DPS data object serializer. Instead of doing the actual serialization, the modified serializer only counts the number of bytes of the data object using the size description of the data structures it contains, without performing any memory copies. Hence, the memory of data structures does not need to be allocated. When partial direct execution is used and the content of the application's data can be ignored, allocating large data structures may be avoided.

Modeling the duration of the individual operations and data object 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 [11]. Since parametric models allow the different performance factors to be isolated from one another, they are very convenient for studying the behavior of a system. One may modify the bandwidth and latency parameters to evaluate the benefits of a faster network, or reduce the duration of various operations to identify the ones that should be optimized. The simulator then becomes a powerful tool for the optimization of parallel applications.

Given the topology of the network connecting the virtual nodes and the state of the current data object transfers, the simulator predicts their completion time by taking network contention into account. The simulator assumes that all incoming, respectively outgoing data transfers for a given node receive an equal share of the available bandwidth. The communication network between the nodes is assumed to have a star topology, where each node has a full duplex link connecting it to a central full crossbar switch which is never a bottleneck.

Since computations and communications may overlap, the processing power needed to handle communications also needs to be taken into account. Receiving data objects induces more inter-

rupts and more memory copies than sending data objects, and is thus more costly. Moreover, we noticed that the consumed processing power depends on the number of outgoing and incoming communications. Since the simulator handles all communications, it knows at every time point how many concurrent transfers are carried out by each processing node, and may thus compute the amount of processing power still available to the concurrently running operations.

We also assume that the processing power not used for communications is shared evenly among all running operations, and that no memory swapping occurs. Similarly to the bandwidth and latency measurements, the required processing power 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.

5. Test application

In order to measure the accuracy of our simulator, we choose a parallel block LU factorization application [8]. The block-based LU factorization relies on the recursive decomposition of the matrix. 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:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix}$$

where A_{11} is a square block of size r x r.

r n-r

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 [12]. 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 the threads in column blocks of size $r \times n$. The flow graph for the LU decomposition is shown in Figure 5. The recursion on the matrix factorization is obtained by replicating a part of the graph (in gray) once for each LU factorization level. The most expensive part of the LU factorization presented here 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 DPS 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 data objects sent to each of the matrix block multiplications (d) contain two matrix blocks of size $r \times r$. The routing function is designed 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).

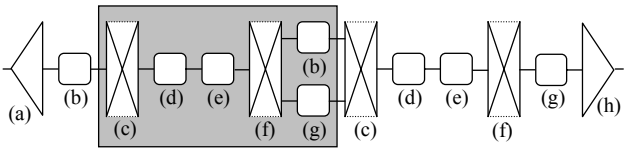

Figure 5. Flow graph for LU factorization. The gray part is repeated for every column of blocks in the matrix. (a) LU factorization of top left block A_{11} (step 1); (b) solve in parallel the triangular system in order to compute T_{12} for all other column blocks and perform row flipping (step 2); (c) collect notifications of finished triangular system solve and stream out multiplication requests; (d) parallel block-based matrix multiplication for $L_{21} \cdot T_{12}$; (e) subtract result of multiplication from B in parallel; (f) collect notifications for end of multiplications (step 3), perform next level LU factorization as soon as first column block is complete, and stream out triangular system solve requests as other column blocks complete; (g) perform row flipping on previous column blocks; (h) collect row exchange notifications for termination.

6. Modifying the application's flow graph

In order to test our simulator in different situations, we explore the impact of varying the decomposition block size, of modifying the LU factorization flow graph and of using the flow control mechanism of DPS.

In the flow graph of Figure 5, the use of the *stream* operations (c) and (f) increases the pipelining of the application, i.e. the number of operations having the potential of running concurrently. Within the pipelined application, *trsm* and *LU* operations are performed simultaneously with matrix multiplications and their associated data transfers. If we replace the *stream* operations by *merge-split* pairs of operations, these act as barrier synchronizations and no pipelining occurs. We refer to this less efficient implementation as the *basic* flow graph, as opposed to the *pipelined* flow graph described in Figure 5.

Varying the block size r used for the decomposition impacts the number of operations, and consequently 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.

DPS threads queue incoming data objects until they can be processed. Sending all multiplication requests at once thus fills the queues and delays the processing of the requests sent by sub-

sequent iterations, thereby reducing the pipelining potential. Applying the flow control capabilities of DPS on the stream operations that generate the multiplication requests limits the number of requests awaiting processing for each iteration, enabling operations belonging to successive iterations to be interleaved, thereby improving pipelining (Figure 6).

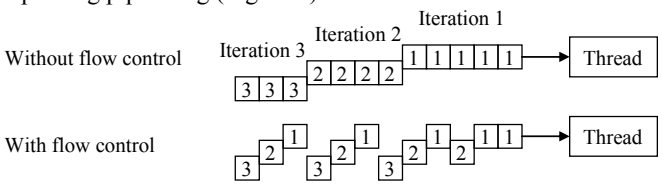

Figure 6. Improved interleaving thanks to the flow control mechanism enables iterations 2 and 3 to be started earlier.

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 7) 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 5 by the flow graph shown in Figure 7.

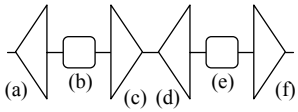

Figure 7. Flow graph for matrix multiplication. (a) Store first matrix, and send column blocks of second matrix; (b) store column blocks; (c) collect notifications; (d) send line blocks of first matrix; (e) multiply received line block with column block stored on thread; (f) collect multiplication results and build resulting matrix.

Since the number of block multiplications decreases at each iteration of the LU factorization, the application processing power requirements decrease over time and the number of allocated nodes may therefore be dynamically reduced. The impact of threads removal on the running time depends on the number of removed threads and on the iteration step of the LU decomposition on which they are removed.

By combining one or several of the proposed modifications and observing their impact on the parallel application's running time, we verify how well the different execution parameters are taken into account by the simulator's network and processing models, and how precisely it reproduces the actual behavior of the parallel application.

7. Improving simulation times and portability through partial direct execution

Let us first present results describing the simulator's performance and portability. 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. 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. This optimized simulator mode is used for all the simulations shown in the next section.

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 2.8GHz (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 reference running time is written in bold.

Table 1 displays simulation results for two different platforms, assessing the portability of our simulator. 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. Therefore, the partial direct execution technique makes the simulation portable without sacrificing accuracy.

8. Validating the simulator

We validate the simulator by comparing the measured and predicted running times of the parallel LU factorization application using the parallelization and pipelining flow graph variations discussed in section 6. All the measurements shown below consider the LU factorization of a 2592x2592 matrix carried out either on four or on eight processing nodes. The machines are Sun workstations with a single 440 MHz UltraSparc II processor connected to a full crossbar switch through a Fast Ethernet network. Hereinafter, we refer to the pipelined flow graph as P, the use of flow control as FC, and to the flow graph with parallel sub-block multiplications as PM. In order to compare the different parallelization strategies, we use the *relative performance improvement* metric, defined as the execution time of the basic flow graph (reference time) over the execution time of the program incorporating one or several of the proposed variations.

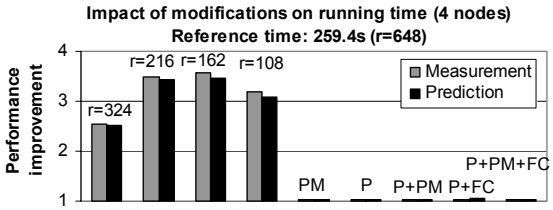

Figure 8. Measured and simulated variation of computation time for the proposed modifications.

In Figure 8, we show the effects of the various optimizations. The reference time (259.4s) is obtained by splitting the matrix in four blocks of 648 columns, distributed on the four available nodes. We see that although the parallel sub-block multiplications (PM), pipelining (P) and flow control (FC) optimizations bring some improvements (around 3%), they are negligible compared with the gains that are obtained by simply changing the decomposition granularity. Splitting the matrix into sixteen blocks (r=162) distributed evenly among the processing nodes yields the shortest measured and predicted running time, respectively 72.5s and 75.5s. The improvement predicted by the simulator is within a few percents of the measured improvements.

Figure 9 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 8. 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 execution instead of accelerating it. On the other hand, pipelining (P) and flow control (FC) slightly improve the performances.

When we increase the number of processing nodes to eight nodes, the pipelined flow graph (P) and the flow control (FC) improvements become more significant (Figure 10). 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.

We now consider the impact of the removal 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 11 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.

This is confirmed by measuring the total execution time of the application for different thread removal strategies (Figure 12). 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, graph "kill 4 after iteration 1"). 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.

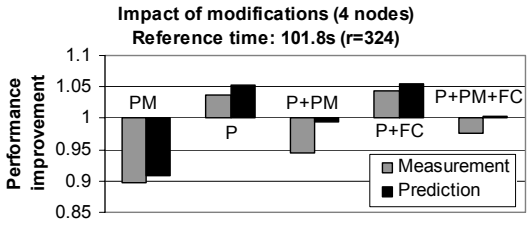

Figure 9. Variation of computation time caused by parallel sub-block multiplications, increased pipelining and flow control. Prediction errors are below 5%.

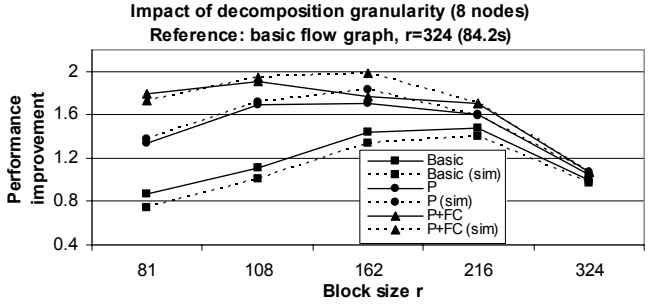

Figure 10. Impact of decomposition granularity on different pipelining strategies.

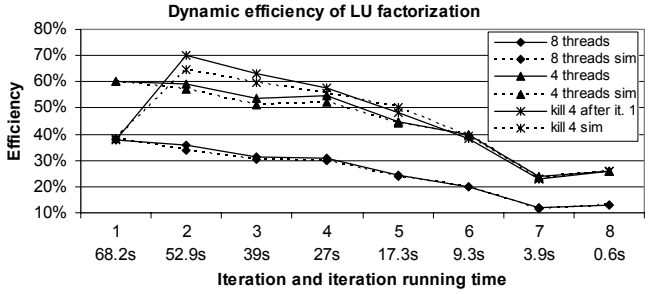

Figure 11. The parallel computation of LU iterations becomes less efficient over time. Removing threads during execution increases the efficiency of the subsequent iterations.

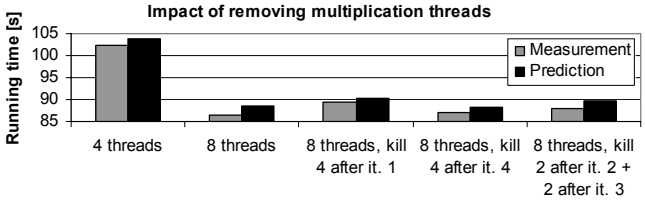

Figure 12. Running times of dynamic thread removal strategies.

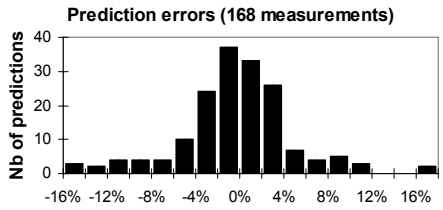

Figure 13. Histogram of prediction errors

In respect to the simulator, Figure 13 shows its prediction accuracy for the 168 measurements carried out for establishing the results shown in the present section. 71.4% of all predictions are within ±4% accuracy, 81.6% are within ±6% accuracy, and more than 95% are within ±12% prediction accuracy.

9. Conclusions and future work

Dynamically allocating and deallocating compute nodes during the execution of parallel applications is a promising technique for improving the utilization of cluster resources. We introduce the concept of dynamic efficiency which expresses the resource utilization efficiency as a function of time. In order to obtain information about the performance and the dynamic efficiency of parallel programs, we propose a simulator built on top of the DPS framework.

In the DPS framework, the parallel structure of an application is specified by a flow graph comprising operations running on DPS threads, routing functions, and data objects moving between operations. The flow graph is constructed at run time and its DPS threads are dynamically deployed onto compute nodes, enabling their dynamic allocation and deallocation.

The extended DPS framework enables the simulation of a DPS application by running all the DPS threads within a single application instance. The simulator coordinates and synchronizes the execution of DPS threads. Operation duration, data transfers, and communication patterns may be derived by direct execution.

The running time, memory requirements and portability of the simulation can be improved by using partial direct execution, i.e. by only executing the parts of the flow graph that send and receive data objects and by predicting the running time of the computations.

We verify the prediction accuracy of our simulator by applying several parallelization strategies to an LU factorization application. 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.

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

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

[6] 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

[7] 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

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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

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

[16] 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

[17] 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