

DPS – Dynamic Parallel Schedules

Sebastian Gerlach, Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne, Switzerland
Sebastian.Gerlach@epfl.ch, RD.Hersch@epfl.ch
<http://dps.epfl.ch>

Abstract

Dynamic Parallel Schedules (DPS) is a high-level framework for developing parallel applications on distributed memory computers (e.g. clusters of PCs). Its model relies on compositional customizable split-compute-merge graphs of operations (directed acyclic flow graphs). The graphs and the mapping of operations to processing nodes are specified dynamically at runtime. DPS applications are pipelined and multithreaded by construction, ensuring a maximal overlap of computations and communications. DPS applications can call parallel services exposed by other DPS applications, enabling the creation of reusable parallel components. The DPS framework relies on a C++ class library. Thanks to its dynamic nature, DPS offers new perspectives for the creation and deployment of parallel applications running on server clusters.

Keywords: *Parallel computation, parallel schedules, flow graphs, split-merge constructs, overlapping of computations and communications*

1. Introduction

The development of high-level tools and languages for simplifying the task of creating parallel applications is one of today's challenges. High-level abstractions should support desirable parallelization patterns. The fundamental "farming" concept of splitting tasks from a master node to worker nodes and gathering the results should be generalized so as to express explicitly the splitting and the merging functions and to be able to map them onto separate computing nodes. Furthermore, since computing clusters built with commodity networks (Fast or Gigabit Ethernet) incur high communication latencies [1] and since an increasing fraction of parallel applications make use of large datasets [2], support should be provided for the overlapping of communications and computations.

A further challenge resides in providing "dynamicity", i.e. allowing parallel programs to modify their behavior and release or acquire resources at run time. Dynamic reshaping of parallel applications at run time is important for parallel server systems whose resources must be reassigned according to the needs of dynamically scheduled applications. And, in order to facilitate the deployment of parallel server systems, parallel programs should be able, at run time, to make use of the services offered by other parallel programs.

High-level parallel programming frameworks for shared memory computers are available mainly for facilitating the development of computational applications and for ensuring the portability of programs [3][4]. In addition, software distributed shared memory systems exist which provide a global shared address space on top of physically distributed memory computers [5].

In the present contribution, we focus on distributed memory systems and try to create abstractions which rely purely on the circulation and distributed processing of data objects. Currently, the majority of parallel application developments running on distributed memory computers are carried out with libraries such as MPI [6] and PVM [7]. These libraries provide low-level message-passing functions, but leave most of the other parallel programming issues to the programmer.

Writing complex programs that execute without deadlocks and make a maximal usage of the underlying hardware such as bi-processor nodes, shared memory for communicating between local threads and overlapping communications and computations requires substantial programming efforts [8]. Low-level message passing libraries also make it difficult to modify parallel programs in order to experiment with different parallel program structures.

Higher-level approaches to parallel application development may provide different levels of abstraction [9]. High abstraction levels such as functional languages leave many decisions related to the program behavior to the parallelization framework and therefore attain only moderate performance. Intermediate abstraction levels, such as CC++ [10], skeletons [11][12][13], and Mentat [14], leave the specification of the parallel program behavior to the programmer but free him from managing communications, threads, synchronization, flow control, and pipelining. They have the potential of facilitating parallel programming and at the same time of achieving a high utilization of the underlying parallel system resources.

Skeleton languages, such as P3L [15][16] or Skil [17] provide pre-implemented parallel constructs. The programmer can combine and customize these constructs by providing his own code. Skeletons exist for task-parallel and data-parallel constructs [18]. Task-parallel constructs are based on the transfer of data items between tasks. Worker tasks process data items as they arrive and send off processed data items as soon as they are ready. Data-parallel constructs work on a distributed data structure that is stored within the worker tasks. Internally, skeletons are usually represented as a tree with nested parallel constructs. Skeleton languages also allow creating sequences of constructs.

We present a new approach to parallel application development, called Dynamic Parallel Schedules (DPS). Dynamic Parallel Schedules have some resemblance with task parallel skeletons, since simple constructs are provided that can be combined and extended by the programmer. Instead of combining the constructs in a tree by nesting them within one another, DPS applications are defined as directed acyclic graphs of constructs¹. This approach allows separating data distribution and collection of results by providing distinct *split* and *merge* constructs. Figure 1 illustrates the flow graph of a simple parallel application, describing the asynchronous flow of data between operations.



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

In contrast to previous approaches [15][17], split and merge operations are extensible constructs, i.e. the developer can provide his own code to control how data is distributed, and how processed sub-results are merged into one result. The data elements in a flow graph are complex data objects. Data objects may contain any combination of simple types or complex types such as arrays or lists. The expressivity of DPS flow graphs is detailed in section 2.

Operations within a flow graph are carried out within threads grouped in thread collections. DPS threads are mapped to operating system threads. Routing of data objects from one operation to the next is accomplished according to user defined routing functions. DPS supports full pipelining of operations. Data objects are transferred as soon as they are computed. Arriving data objects are stored in queues associated with the thread that contains the operations that will process them. This macro data flow behavior enables automatic overlapping of communications and computations. The execution of a flow graph within its collection of threads and according to its routing functions is referred to as a *parallel schedule*.

For solving data-parallel problems, operations can store data within their local threads, e.g. a matrix distributed across different nodes. Libraries of flow graphs can be created to perform operations on distributed data structures. These flow graphs can then be used by applications for performing higher level computations.

The present approach to parallel application development was first introduced in the context of data-intensive computing applications. The first generation parallel schedule system successfully performed out-of-core parallel access to 3D volume images [20], computation of radio-listening rates [21], and streaming real-time slice extraction from a time-varying 3D volume image [22]. These applications allowed to validate the approach and to identify the desirable new features of the second generation DPS framework. These new features include facilities

for simplifying the creation of parallel schedules, for providing interoperable parallel services and for allowing the dynamic deployment of applications.

Therefore all DPS structures that describe the application such as its flow graph and thread mapping are created dynamically at runtime. This dynamic behavior allows applications to reconfigure themselves in order to adapt to changes in the problem definition or in the computing environment without requiring recompilation or restarting. A DPS application can expose its graphs to other DPS applications, thus enabling a parallel application to call parallel services exposed by another parallel application. Finally, the second generation framework adds the *stream* construct to the model. The stream construct enables partial merging and forwarding of data elements in order to ensure the pipelining of subsequent sets of parallel operations (section 3).

High-level approaches for parallel programming often rely on a new programming language, or add extensions to existing languages. In order to avoid requiring program developers to learn a new language or language extension, DPS applications are written in pure C++. The parallel constructs are handled in DPS by using C++ classes, templates, macros and operator overloading.

DPS programs may be recompiled and run without modification on platforms on which the DPS library has been ported (presently Windows and Linux).

The remainder of this paper is structured as follows. Section 2 presents the basic concepts for creating DPS schedules. Section 3 describes the functionality and interfaces of the DPS C++ library. Section 4 gives an overview of the runtime system and of some of its associated performance issues. Section 5 presents two applications illustrating the philosophy and the concepts of DPS. Section 6 draws the conclusions and presents future research directions.

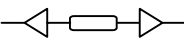
2. Expressing Parallel Schedules

An application using Dynamic Parallel Schedules is expressed as a directed acyclic graph of sequential operations. The nodes on the graph are user-written functions deriving from the elementary DPS operations: *leaf* operation, *split* operation, *merge* operation, and *stream* operation. Pipelining of operations is implicit in the construction of the flow graph.

The basic parallel construct (split, computation, merge) is illustrated in Figure 1. The split operation takes as input one data object, and generates as output multiple output data objects representing the subtasks to execute. Each *ComputeData* leaf operation processes the data of the incoming data object and generates one output data object. The merge operation collects all the output data objects to produce one global result. The programmer does not have to know how many data objects arrive at the merge operation, since DPS keeps track of the number of data objects generated by the corresponding split operation. An additional operation, the *stream* operation, is described in section 3.

The flow graph shown in Figure 1 is an unfolded view of a DPS graph showing its inherent parallelism. However, DPS graphs are generally represented by only single instances of

¹ Task graphs [19] also describe parallel applications as directed acyclic graphs of sequential operations. They however do not offer a split-merge construct, a fundamental element facilitating the creation of parallel programs.

parallel constructs, e.g.  for a split-computation-merge construct.

As it stands, a flow graph only indicates the processing operations and their order of execution. To enable parallelism, the operations need to be mapped to different *threads*, possibly located in different processing nodes. Threads may incorporate user defined data structures. They are associated to operating system threads and are mapped onto the nodes where their operations will execute. Developers instantiate collections of threads. A user-defined routing function specifies at runtime to which instance of the thread in the thread collection a data object is directed in order to execute its next operation.

The execution of a sequence of leaf operations within a flow graph is automatically pipelined. This enables overlapping of computations, communications and possibly I/O operations.

The graph elements are compositional: a split-merge construct may contain another split-merge construct. A split-merge construct may incorporate different paths, enabling a data-dependant conditional execution of parts of the flow graph at runtime.

DPS flow graphs and the mapping of thread collections to operating system threads are specified dynamically at run time by the application. Dynamic flow graphs enable to adjust the graph to the size of the problem to be solved. In section 5, we show how a dynamic graph is used to parallelize the LU factorization. Dynamically created thread collections and mappings of threads to nodes also offer the potential for dynamically allocating computing and I/O resources according to the requirements of multiple concurrently running parallel applications.

3. DPS library constructs

The DPS C++ library provides a class framework for developing parallel applications. For maximum ease of use and maintainability, it does not extend the C++ language in any way. DPS applications can be compiled with a standard C++ compiler such as gcc or MS Visual C++. To simplify the development and debugging effort, the library takes care of serialization and deserialization of the data objects used in the application without requiring additional coding. It also detects invalid constructs at compile time, such as an attempt to create a sequence of operations in a flow graph where the output data object type is different from the input data object type of the next operation. The library also takes care of releasing memory using smart pointers with reference counting.

The following paragraphs illustrate the syntax of the various DPS constructs. The source code originates from a tutorial application serving as a first introduction to DPS. It converts in parallel a character string from lowercase to uppercase by splitting the string into its individual character components.

Expressing data objects

Data objects are the fundamental data elements used in DPS. A data object is a standard C++ class, with additional information enabling serialization and deserialization. The following program lines describe a simple data object.

```
class CharToken : public SimpleToken {
public:
```

```
    char chr; // a character
    int pos; // its position within a string
    // Constructor for CharToken
    CharToken(char c = 0, int p = 0) { chr=c; pos=p; }
    IDENTIFY (CharToken);
};
```

The elements added for DPS is the *identify* macro. This macro provides support for serialization, deserialization, and to create an abstract class factory to instantiate the data object during deserialization [23]. The *CharToken* simple data object type is serialized and deserialized with simple memory copies. More complex data object types can also be created, as illustrated in the following example.

```
class MyComplexToken : public ComplexToken {
public:
    CT<int> id; // A simple integer
    CT<std::string> name; // A character string
    // A vector of Somethings
    Vector<Something> children;
    // A variable-size array of integers
    Buffer<int> aBuffer;
    IDENTIFY (MyComplexToken);
};
```

This data object contains complex data types and containers. DPS provides two container templates that can store variable-size arrays of simple elements (*Buffer*) or complex elements (*Vector*). Complex data objects can only contain complex elements; therefore a templated class (*CT*) is provided for inserting simple types or types otherwise known to the serializer, such as *std::string*. These templates enable programmers to create complicated data structures within their data objects without having to care about serialization. Support for derived classes is also provided. The serialization is performed with pointer arithmetic in order to traverse the elements of the data object. The present approach enables serialization without requiring redundant data declarations.

Expressing operations

Operations are also expressed as C++ classes deriving from DPS provided base classes. The following source code shows the three operations of a simple split-compute-merge graph. *SplitString* and *MergeString* operations are executed by an instance of *MainThread*. The *ToUpperCase* leaf operations are performed by instances of *ComputeThread*.

```
class SplitString :
    public SplitOperation<MainThread, // Thread
        TV1(StringToken), TV1(CharToken)> {
    // Input token types Output token types
    // TV1 indicates one argument type
public:
    void execute(StringToken *in) {
        // Post one token for each character
        for(int i=0;i<STRLEN;i++)
            postToken(new CharToken(in->str[i],i));
    }
    IDENTIFYOPERATION(SplitString);
};

class ToUpperCase :
    public LeafOperation<ComputeThread,
        TV1(CharToken), TV1(CharToken)> {
public:
    void execute(CharToken *in) {
```

```

// Post the uppercase equivalent
// of the incoming character
postToken(
    new CharToken(toupper(in->chr), in->pos));
}
IDENTIFYOPERATION (ToUpperCase);
};

class MergeString :
    public MergeOperation<MainThread,
        TV1(CharToken), TV1(StringToken)> {
public:
void execute(CharToken *in) {
    StringToken *out=new StringToken();
    do {
        // Store incoming characters at
        // the appropriate position of string
        out->str[in->pos]=in->chr;
    }
    // Wait for all chars
    while(in=(CharToken*) (Token*)
        waitForNextToken());
    // Post output string
    postToken(out);
}
IDENTIFYOPERATION (MergeString);
};

```

As with data objects, the operations have *identify* macros. The template parameters for the base classes indicate the thread class with which the operation is associated, and the acceptable input and output data object types. These parameters are used for verifying the graph coherence at compile time.

Expressing threads and routing functions

Threads are also expressed as classes. They can contain members for storing data elements in order to support the construction of distributed data structures. The following source code shows a simple thread type containing a single member variable.

```

class ComputeThread : public Thread {
    int threadMember;
    IDENTIFY (ComputeThread);
};

```

Routing functions are also expressed as classes, as shown in the following lines of code:

```

class RoundRobinRoute :
    public Route<ComputeThread, CharToken> {
    // Target thread type Token type
    int route(CharToken *currentToken) {
        // Return a thread index
        return currentToken->pos%threadCount();
    }
    IDENTIFY (RoundRobinRoute);
};

```

The route function contains an expression returning an index to one thread in a thread collection. Due to the simplicity of most routing functions, a ROUTE macro is provided. The parameters are the name of the routing function, the associated thread type, the data object type to be routed, and the routing expression producing the destination thread index. The following ROUTE macro produces the same routing function as the one described above:

```

ROUTE (RoundRobinRoute, ComputeThread,
    CharToken, currentToken->pos%threadCount());

```

Expressing thread collections and flow graphs

With all static elements of an application defined, the dynamic construction of thread collections and flow graphs can now be described. Thread collections are simply instantiated and named:

```

Ptr<ThreadCollection<ComputeThread> >
computeThreads =
    new ThreadCollection<ComputeThread>("proc");

```

The mapping of the threads of a thread collection onto nodes is specified by using a string containing the names of the nodes separated by spaces, with an optional multiplier to create multiple threads on the same node. This string can be loaded from a configuration file, be specified as a constant, or be created at runtime, according to the application's requirements. The following lines show the simplest form, where a constant is specified to create three threads, two on node *nodeA*, and one on node *nodeB*.

```

computeThreads->map("nodeA*2 nodeB");
// The string specifies the mapping

```

Flow graphs are defined with overloaded C++ operators. The following lines of source code can be used to create a flow graph (Figure 2) containing the three previously defined operations (split, computation, and merge operations). The flow graphs are named in order to possibly reuse them by other applications.

```

FlowgraphBuilder theGraphBuilder =
    FlowgraphNode<SplitString, MainRoute>
        ( theMainThread ) >>
    FlowgraphNode<ToUpperCase, RoundRobinRoute>
        ( computeThreads ) >>
    FlowgraphNode<MergeString, MainRoute>
        // Operation Routing func
        ( theMainThread );
        // Thread collection

Ptr<Flowgraph> theGraph=new Flowgraph
    (theGraphBuilder,"graph");
// Builder name of graph

```

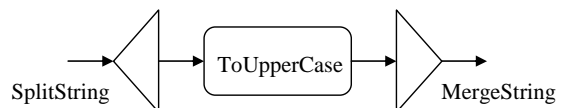


Figure 2. Simple flow graph

The flow graph nodes represented by *FlowgraphNode* objects specify the operation, the route to this operation and the thread collection on which the operation should execute. The operator >> is used to indicate paths in the graph. The operator >> generates compile time errors when two incompatible operations are linked together (e.g. when their data object types do not match).

More complex graphs are created by declaring *FlowgraphNode* variables and reusing them to create separate paths. For instance, in order to create a graph with two possible different types of operations between the split and merge operations (Figure 3), we can use the following lines of source code.

```

FlowGraphNode<MySplit, MainRoute>
    nodeSplit (theMainThread);
FlowGraphNode<MyMerge, MainRoute>
    nodeMerge (theMainThread);
FlowGraphNode<MyOpOne, RoundRobinRoute>
    nodeOp1 (ComputeThreads);
FlowGraphNode<MyOpTwo, RoundRobinRoute>
    nodeOp2 (ComputeThreads);

// create 1st path in graph
FlowgraphBuilder theGraphBuilder =
    nodeSplit >> nodeOp1 >> nodeMerge;
// add 2nd path to graph
theGraphBuilder +=
    nodeSplit >> nodeOp2 >> nodeMerge;

Ptr<Flowgraph> theGraph = new Flowgraph
    (theGraphBuilder, "graph");

```

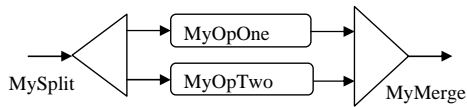


Figure 3. Graph with two possible paths; the selected path depends on the data object type.

Note that the += operator allows to insert an additional path to the graph. It can also be used to append pieces of graphs together, e.g. to create a graph of varying length, as illustrated in the LU factorization example (section 5). When multiple paths are available to a given output data object, the input data object types of the destinations are used to determine which path to follow. In the example of Figure 3, *MyOpOne* and *MyOpTwo* must have different input data object types. Programmers may create at runtime different types of data objects that will be routed to different operations.

Stream operations

In the previous paragraphs we presented graphs containing split, leaf, and merge operations. DPS offers in addition the *stream* operation.

In some applications, it may be useful to collect data objects as in a merge operation, but to post more than one output data object. This may be carried out by using a sequence comprising a merge and a split operation, but no output data objects would be posted before the merge received all its input data objects. To enable pipelining, DPS offers the *stream* construct. It works like a merge and a split operation combined, enabling the programmer to post data objects at any appropriate time during the execution of the operation. A graph using the stream operation in a simple video processing application is illustrated in Figure 4. An uncompressed video stream is stored on a disk array as partial frames, which need to be recomposed before further processing. The use of the stream operation enables complete frames to be processed as soon as they are ready, without waiting until all partial frames have been read. Another application for the stream operation is shown in the LU factorization example (Section 5).

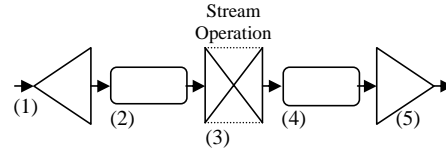


Figure 4. Graph with *stream* operation for processing video: (1) generate frame part read requests; (2) read frame parts from the disk array; (3) combine frame parts into complete frames and stream them out; (4) process complete frames; (5) merge processed frames onto the final stream.

Flow control and load balancing

Since DPS tracks the data objects travelling between split/merge pairs, a feedback mechanism ensures that no more than a given number of data objects is in circulation between a specific pair of split/merge constructs. This prevents the split operation from sending many data objects in a very short time interval, which would possibly induce a very high memory or network load. The split operation is simply stalled until data objects have arrived and been processed by the corresponding merge operation. By incorporating additional information into posted data objects, such as the processing nodes to which they were sent, DPS achieves a simple form of load balancing. After the split operation, the routing function sends data objects to those processing nodes which have previously posted data objects to the merge operation. Such a scheme allows balancing the load within the nodes spanned by a split-merge construct.

Sequencing of operations

At the heart of the DPS library is the *Controller* object, instantiated in each node and responsible for sequencing within each node the program execution according to the flow graphs and thread collections instantiated by the application. The controller object establishes all required connections, creates threads, and is responsible for the transmission of the flow graph and the thread collection information to newly launched application node instances.

4. Runtime Support

The DPS runtime environment for a typical usage case is illustrated in Figure 5. DPS provides a kernel that is running on all computers participating in the parallel program execution. This kernel is used for launching parallel applications and for initiating communications within a parallel application or between two distinct parallel applications. A running application may use the services provided by another running application by calling its flow graphs.

The kernels are named independently of the underlying host names. This allows multiple kernels to be executed on a single host. This feature is mainly useful for debugging purposes. It enforces the use of the networking code (serialization/deserialization) and of the complete runtime system although the application is running within a single computer.

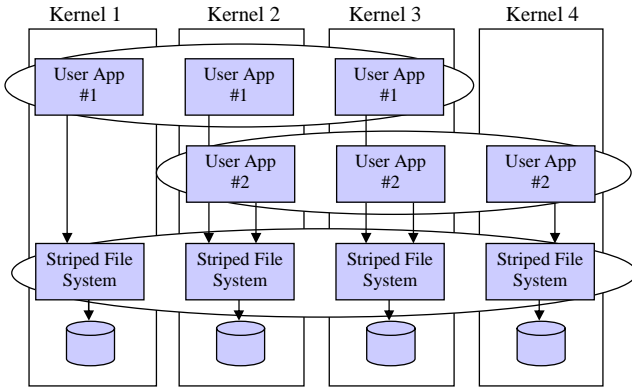


Figure 5. Two parallel applications calling parallel striped file services provided by a third parallel application within the DPS runtime environment

The DPS runtime system was designed to be as dynamic as possible. The kernels can be started or stopped at any point in time to add or remove nodes from the cluster. Kernels locate each other either by using UDP broadcasts or by accessing a simple name server. When an application is started on a given node, it first contacts the local kernel, and starts constructing its flow graphs and thread collections. DPS uses a delayed mechanism for starting communications. It neither launches an application on a node nor opens a connection (TCP socket) to another application unless a data object needs to reach that node. When an application thread posts a data object to a thread running on a node where there is no active instance of the application, the kernel on that node starts a new instance of the application. This strategy minimizes resource consumption and enables dynamic mapping of threads to processing nodes at runtime. However, this approach requires a slightly longer startup time (e.g. one second on an 8 node system), especially for applications that need full N-to-N node connectivity.

DPS performs communications using TCP sockets. When a data object is sent between two threads within the same address space, it bypasses the communication layer – the pointer to the data object is transferred directly to the destination thread. Thus messages are transferred at a negligible cost between threads of a shared memory multiprocessor node.

Communication overhead

The communication overhead of DPS was evaluated with several simple experiments. These experiments were executed and timed on a cluster of bi-processor 733MHz Pentium III PCs with 512 MB of RAM, running Windows 2000. The cluster is composed of 8 computers (nodes), interconnected with a Gigabit Ethernet switch.

In order to evaluate the maximal data throughput when performing simultaneous send and receive operations, the first test transfers 100 MB of data along a ring of 4 PCs. The individual machines forward the data as soon as they receive it. In Figure 6, we compare the steady state data transfer throughput through the four computing nodes by receiving and sending blocks (a) directly through a socket interface and (b) by embedding data of the same size into DPS data objects.

Data objects transferred over the network incorporate control structures giving information about their state and position within the flow graph. These control structures induce an overhead that is significant only when sending large amounts of small data objects.

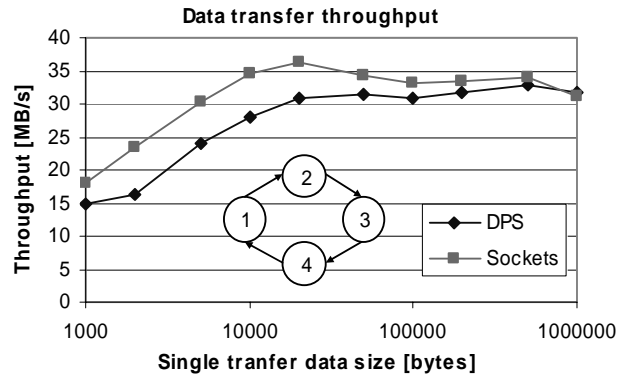


Figure 6. Round-trip data transfer throughput: comparing DPS with transfers relying on direct socket accesses

Benefits of overlapping communications and computations

The second experiment illustrates the benefits of the implicit overlapping of communications and computations obtained with DPS graphs. To evaluate this overlap, we run a program multiplying two square $n \times n$ matrices by performing block-based matrix multiplications. Assuming that the $n \times n$ matrix is split into s blocks horizontally and vertically, the amount of communication is proportional to $n^2 \cdot (2s+1)$, whereas computation is proportional to n^3 . By keeping the size of the matrix n constant and varying the splitting factor s , the ratio between communication time and computation time can be modified. For this test, two 1024×1024 element matrices are multiplied on 1 to 4 compute nodes, with block sizes ranging from 256×256 ($s=4$) to 32×32 ($s=32$). This enables testing situations where either communications ($s=16$ and $s=32$) or computations ($s=4$ and $s=8$) are the bottleneck. The reductions in execution time due to overlapping of communications and computations and the corresponding ratios of communication time over computation time are given in Table 1.

Nodes	Block size 256		128		64		32	
	reduct.	ratio						
1	6.7%	0.22	9.1%	0.45	17.6%	0.94	25.2%	2.09
2	13.6%	0.33	19.8%	0.66	28.7%	1.28	24.9%	2.76
3	15.8%	0.44	29.5%	0.97	32.1%	1.92	19.5%	4.19
4	23.9%	0.63	35.6%	1.36	27.2%	2.54	15.6%	5.54

Table 1. Reduction in execution time due to overlapping and corresponding ratio of communication over computation time

The potential reduction g in execution time due to pipelining is either

$$g = \text{ratio}/(\text{ratio}+1) \quad \text{if ratio} \leq 1, \text{ or}$$

$$g = 1/(1+\text{ratio}) \quad \text{if ratio} \geq 1.$$

Potential and measured reductions in execution time are the closest when the communication over computation time ratio is high, i.e. higher than 90%. This is easily explained by the fact that when communication dominates, processors tend to be partially idle. The highest gains in execution time are obtained at ratios of communication over computation times between 0.9 and 2.5. Out of a maximum of 50%, Table 1 shows that DPS automatic pipelining yields execution time reductions between 25% and 35% when communication time is similar or up to 2.5 times higher than computation time.

5. Application examples

In order to evaluate the functionality and measure the performances obtained under DPS, several ‘traditional’ parallel applications have been developed. Here, we present the parallelization of the game of life and of the LU matrix factorization. These examples illustrate the use of DPS constructs for non-trivial parallelization problems. The configuration of the PC cluster is the same as described in the previous section.

Game of Life

The parallel implementation of Conway’s Game of Life is especially interesting since it exhibits a parallel program structure similar to many iterative finite difference computational problems [24].

The world data structure is evenly distributed between the nodes, each node holding a horizontal band of the world. Each computation requires knowledge of the state of lines of cells held on neighboring nodes. A simple approach consists in first exchanging borders, and after a global synchronization, computing the future state of the world. The corresponding DPS flow graph is illustrated in Figure 7.

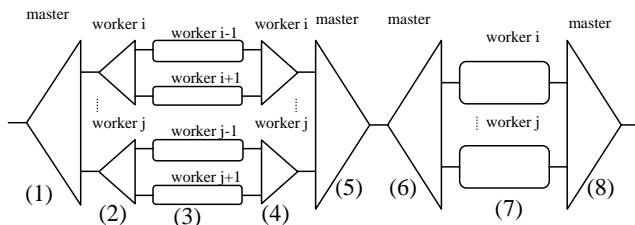


Figure 7. Simple flow graph for the parallel game of life (unfolded view): (1) split to worker nodes; (2) split border transfer request to neighboring nodes; (3) neighbors send the borders; (4) collect borders; (5) global synchronization to ensure that all borders have been exchanged; (6) split computation requests; (7) compute next state of world; (8) synchronize end of current iteration.

The computation of the future state of the center of the part of the world stored on a node can be carried out without knowledge of any cell lines located on the neighboring nodes. We can perform this computation in parallel with the border exchange. A new flow graph (Figure 8) can thus be constructed, by keeping most of the operations as they were in the previous graph.

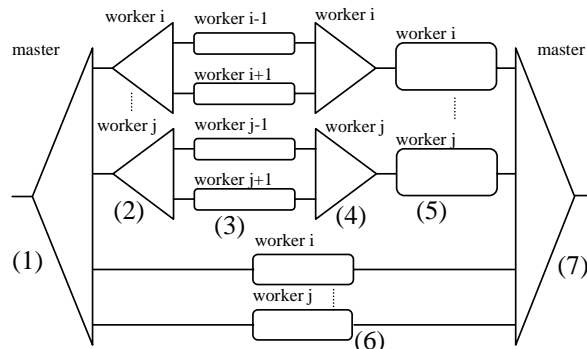


Figure 8. Improved flow graph for the parallel game of life (unfolded view): (1) split to worker nodes; (2) split border transfer request to neighboring nodes; (3) neighbors send the borders; (4) collect borders; (5) compute next state of borders; (6) compute next state of center; (7) synchronize end of current iteration.

Figure 9 shows the relative performances for both configurations, as a function of world size. In all cases, the improved approach yields a higher performance. With the smallest world size, the communications overhead is the largest and the difference between the two approaches is the most pronounced. Larger world sizes reduce the impact of communications and therefore the potential gain of carrying out computations and exchange of borders in parallel.

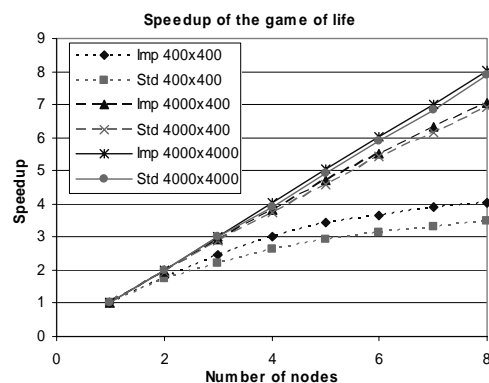


Figure 9. Speedup for the game of life, improved versus simple flow graph, for different world sizes

Exposing the Game of Life as a parallel service

To illustrate the parallel service capabilities of DPS, the game of life has been extended by providing an additional graph that returns the current state of a subset of the world, possibly distributed over several compute nodes. A visualization application interacts with the game of life by calling this graph to display the world as it evolves (Figure 10).

The client graph calls the graph exposed by the game of life. It is seen by the client application as a simple leaf operation. Thus pipelining and data object queuing is preserved when carrying out such calls.

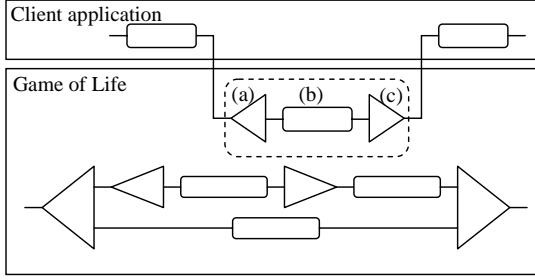


Figure 10. Inter-application graph call. The client calls a graph exposed by the game of life. (a) split request to worker nodes; (b) read requested parts; (c) merge parts into desired subset of the world

To measure the overhead of graph calls, the client application periodically requests randomly located fixed-sized blocks from a world of 5620x5620 cells. When running on 4 machines without visualization graph calls, calculating one iteration takes 1000 ms. Table 2 shows the impact of the graph calls on the simulation speed. The call time is divided into processing time (reading the world data from memory) and communication time. The implicit overlap of communications and computations enables graph calls to be executed very efficiently.

Block size		Time per call (median)	Simulation iteration time	Average number of calls/sec
width	height			
			1000 ms	None
40	40	1.66 ms	1041 ms	66.8
400	400	22.14 ms	1284 ms	31.8
400	2400	130.43 ms	1381 ms	6.9

Table 2. Simulation iteration time with and without graph calls

LU Factorization

Block LU factorization with partial pivoting [25] is an interesting case for parallelization, since it incorporates many data dependencies. The block-based LU factorization was chosen since it produces many matrix multiplications, which can be easily distributed to all participating nodes. To better understand the DPS graph, let us quickly review the process of block LU factorization. We split the matrix A of size $n \times n$ that we intend to factorize into 4 blocks.

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

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.

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \cdot U_{11} \quad \text{where } L_{11} \text{ is a lower triangular matrix and } U_{11} \text{ is an upper triangular matrix.}$$

Step 2. Compute T_{12} by solving the triangular system. This is the operation performed by the *trsm* routine in BLAS [26]. Carry out row flipping according to the partial pivoting of step 1.

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

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 recursively apply the block LU factorization until A' is a square matrix of size r .

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

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

To carry out the LU factorizations of very large matrices, we distribute the matrix to factorize onto the computation nodes as columns of vertically adjacent blocks. The corresponding matrix operations are shown in Figure 11, and the graph is illustrated in Figure 12.

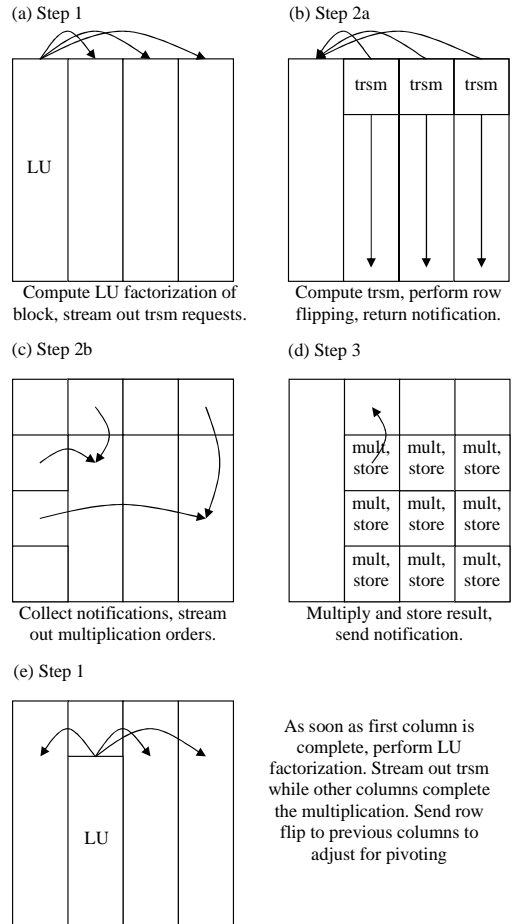


Figure 11. Operations in graph for LU factorization

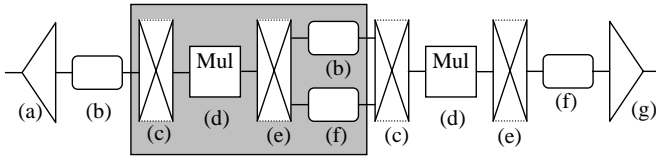


Figure 12. Graph for LU factorization. The gray part is repeated for every column of blocks in the matrix. (a) LU factorization of top left block (step 1) and split to columns; (b) solve triangular system for all other columns and perform row flipping (step 2a); (c) collect notification of finished triangular system solves and stream out multiplications (step 2b); (d) matrix multiply (step 3); (e) collect notifications for end of multiplications, perform next level LU factorization as soon as first column is complete, and stream out triangular system solves as other columns complete; (f) perform row exchange on previous columns; (g) collect row exchange notifications for termination.

Figure 13 illustrates the unfolded graph for a matrix that is split into 4 by 4 blocks. The multiply, trsm and row flip operations are performed in parallel. Thanks to the stream operations, processing can advance further into the graph before all trsms or multiplications for a given step are complete, thus ensuring pipelining within the execution of the application.

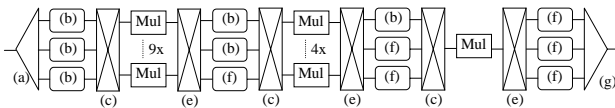


Figure 13. Unfolded graph for the LU factorization of a matrix subdivided into 4x4 blocks, performed according to Figure 12

In the above graphs, the multiplication is represented by a simple box. It is itself a split-merge construct, as illustrated in Figure 14. Parallel operations (b) are used to collect the operands of each multiplication. The subsequent matrix block multiplication is performed within the merge operation (c), which, for load balancing purposes, is carried out in a separate thread collection. A separate operation (d) transfers the result onto the node where it is needed for further processing. This example illustrates the capabilities of DPS split-merge constructs for specifying and executing collective data gathering, processing, and relocation operations.

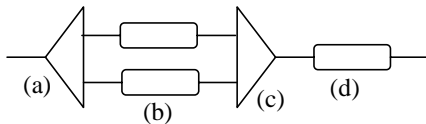


Figure 14. Matrix multiplication within the LU factorization: (a) split operation; (b) collect both operands; (c) multiply both operands; (d) store the result in the target thread local storage

The LU factorization shows the benefits of dynamically created graphs. The graph is created to fit the size of the problem. It also illustrates the approach a developer takes when parallelizing complex problems – the graph reflects the data flow in the application. The developer must ensure that the individual

DPS blocks behave as expected, and DPS takes care of all pipelining, synchronization and scheduling issues.

Figure 15 shows the performance of the LU factorization with a matrix of 4096x4096 elements. No optimized linear algebra library was used for this implementation. The graph shows the relative performance of two variants: the first one is fully pipelined, and the second one uses a standard merge-split construct instead of the stream operations. It clearly illustrates the additional performance gain obtained thanks to the pipelining offered by the *stream* operations.

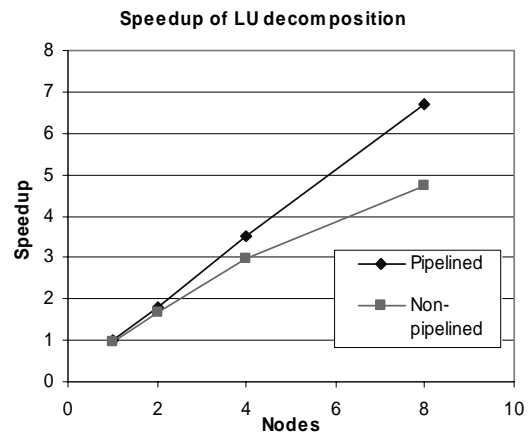


Figure 15. Performance of LU factorization

6. Conclusions and Future Work

DPS is a novel environment for the creation of parallel programs. DPS generalizes the “farming” concept by allowing *split* operations (distribution of “tasks” to workers) and *merge* operations (collection of results) to incorporate application-specific code and by allowing them to be mapped onto distinct nodes. Furthermore, the *stream* operation allows to pipeline two successive split-merge constructs. DPS applications are specified by a flow graph representing hierarchies and sequences of split, computation, merge and stream operations. The flow graph (acyclic directed graph) represents the parallel program execution pattern. It can be easily visualized and represents therefore a valuable tool for thinking and experimenting with different parallelization strategies.

DPS applications are by construction automatically multithreaded and pipelined, thus yielding overlapped computations and communications.

The runtime environment and the library enable the specification of flow graphs and the mapping of threads to nodes at runtime. This opens up possibilities for dynamic resource allocation and for load balancing between different applications scheduled within a single cluster. This is of particular interest in server environments, where various services need to be provided with a limited set of resources and under continuously evolving load profiles. The ability to call graphs exposed by other DPS applications enables the development of complex programs split into smaller reusable components.

In the near future, we intend to make the inter-application graph building capacity more flexible by allowing corresponding split and merge operations to reside in different applications. Inter-application split and merge operations are the key to interoperable parallel program components. They allow a server application having knowledge about the distribution of data, to serve a request to access in parallel many data items by performing a split operation. The client application may then directly process the data items in parallel and combine them into a useful result by performing a merge operation.

Within our research on multimedia servers, we will also explore the possibility of allocating additional resources to a running program at runtime by taking advantage of the dynamicity of DPS. The dynamicity of DPS combined with appropriate checkpointing procedures may also lead to more lightweight approaches for graceful degradation in case of node failures.

To allow a wider use of DPS, the software is available on the Web under the GPL license at <http://dps.epfl.ch>.

7. Acknowledgements

We would like to thank Benoit Gennart for having introduced the concept of parallel schedules and Marc Mazzariol for having contributed to the initial specification of dynamic parallel schedules.

References

- [1] M. Lobosco, V. Santos Costa, C. Luis de Amorim, Performance Evaluation of Fast Ethernet, Giganet, and Myrinet on a Cluster. Int'l Conf on Computational Science (ICCS2002), P.M.A. Sloot et al. (Eds.), LNCS 2329, Springer, pp. 296-305, 2002
- [2] J. Saltz, A. Sussman, S. Graham, J. Demmel, S. Baden, J. Dongarra, Programming Tools and Environments, Communications of the ACM, Vol. 41, No. 11, pp. 64-73, 1998
- [3] C. Koelbl, D. Loveman, R. Schreiber, G. Steele, M. Zosel, The High Performance Fortran Handbook, MIT Press, 1994
- [4] The OpenMP Forum, OpenMP C++ Applications Program Interface, <http://www.openmp.org>, Oct. 2002
- [5] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwanenpoel, ThreadMarks: Shared memory computing on networks of workstations, IEEE Computer, Vol 29, No. 2, 18-28, Feb. 1996
- [6] J. Dongarra, S. Otto, M. Snir, D. Walker, A message passing standard for MPP and Workstations, Communications of the ACM Vol. 39, No. 7, pp. 84-90, 1996
- [7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994
- [8] S. Baden, S.S. Fink, A programming methodology for dual-tier multicomputers, IEEE Transactions on Software Engineering, Vol. 26, No. 3, pp. 212-226, March 2000
- [9] D. Skillicorn, D. Talia, Models and Languages for Parallel Computation, ACM Computing Surveys, Vol. 30, No. 2, pp. 123-169, June 1998
- [10] K. M. Chandy, C. Kesselman, CC++: A Declarative Concurrent Object Oriented Programming Notation, Research Direction in Concurrent Object-Oriented Programming, MIT Press, pp. 281-313, 1993
- [11] H. Kuchen, A Skeleton Library, Proc. Euro-Par 2002, LNCS 2400, Springer-Verlag, pp. 620-629, 2002
- [12] M. I. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, 1989
- [13] J. Darlington, Y. Guo, H. W. To J. Yang, Parallel Skeletons for Structured Composition, Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 19-28, 1995
- [14] A. S. Grimshaw, Easy-to-Use Object-Oriented Parallel Processing with Mentat, IEEE Computer, Vol. 26 No. 5, pp. 39-51, 1993
- [15] B. Bacci, M. Danuletto, S. Orlando, S. Pelagatti, M. Vanneschi, P3L: a Structured High-level Parallel Language, and its Structured Support, Concurrency Practice and Experience, 7(3), pp. 225-255, May 1995
- [16] S. Ciarpaglini, L. Folchi, S. Pelagatti, Anacleto: User manual, Nov. 1998, <http://www.di.unipi.it/~susanna/p3longoing.html>
- [17] G. H. Botorog, H. Kuchen, Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, Proc. 5th International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, pp. 243-252, 1996
- [18] H. Kuchen, M. I. Cole, The Integration of Task and Data Parallel Skeletons, Proc. of the 3rd International Workshop on Constructive Methods for Parallel Programming 2002, TU Berlin, Forschungsberichte der Fakultät IV, No. 2002/07, ISSN 1436-9915, pp. 3-16, 2002
- [19] D. B. Skillicorn, The Network of Tasks Model, Proc. of Parallel and Distributed Computing Systems 1999, IASTED, available as Report 1999-427, Queen's University School of Computing, <http://www.cs.queensu.ca/TechReports/authorsS.html>
- [20] V. Messerli, O. Figueiredo, B. Gennart, R. D. Hersch, Parallelizing I/O intensive Image Access and Processing Applications, IEEE Concurrency, Vol. 7, No. 2, pp. 28-37, April-June 1999
- [21] M. Mazzariol, B. Gennart, R.D. Hersch, M. Gomez, P. Balsiger, F. Pellandini, M. Leder, D. Wüthrich, J. Feitknecht, Parallel Computation of Radio Listening Rates, Proc. Conf. Parallel and Distributed Methods for Image Processing IV, SPIE Vol 4118, pp. 146-153, July 2000
- [22] J. Tarraga, V. Messerli, O. Figueiredo, B. Gennart, R.D. Hersch, Parallelization of Continuous Media Applications: the 4D Beating Heart Slice Server, Proc. ACM Multimedia, pp. 431-441, 1999
- [23] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, pp. 87-95, 1995
- [24] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, Benjamin Cummings Publishing Company, Chapter 11, Solving sparse systems of linear equations, pp. 407-489, 1993
- [25] G. H. Golub, C. F. van Loan, Matrix Computations, The Johns Hopkins University Press, pp. 94-116, 1996
- [26] 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