Introduction

Writing parallel applications with DPS

Dynamic Parallel Schedules (DPS) is a high-level framework for developing parallel applications on clusters of workstations. An application using DPS is expressed as a directed acyclic graph of sequential operations, called a flow graph. The flow graph is composed of compositional customizable split – process – merge constructs. 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. The DPS framework provides many additional features, such as support for heterogeneous clusters, cross-platform applications, parallel components, malleable applications, and fault-tolerance. This chapter presents the high-level concepts used by DPS.

Flow graph representation

DPS applications are described by their flow graph, which specifies sequences of operations passing data objects from one operation to the next. Consider the following simple sequential C program:

```c
fread(inBuffer,inBufferSize,1,inputFile);
processData(inBuffer,inBufferSize,outBuffer,outBufferSize);
```

The first instruction reads data from a file to a newly allocated memory buffer, and the second instruction processes the data, storing the result in a second buffer. In a data-driven representation of the application, the data elements appearing in the program drive its execution, as shown in Figure 1. The ReadFileRequest data object causes the execution of the ReadFile operation, which produces a DataBuffer output data object. The output data object of ReadFile is subsequently processed by ProcessData, which produces another DataBuffer output data object containing the results of the processing. The involved data objects contain the same information as the parameters passed to the functions in the previous pseudocode.

![Flow graph for a simple application](image)

The parallel execution is described by the graph of execution paths that can be followed by the data objects, the flow graph. The nodes on the graph are the operations that are executed, using a data object as input, and producing a new data object as output. One execution of the flow graph is called a parallel schedule.

Split-Process-Merge flow graphs

The previous example showed a simple sequential application, but the model can easily be extended to parallel applications by using special operations that can produce multiple output data objects or consume multiple input data objects. The flow graph in Figure 2 shows a simple parallel application.

![A simple parallel application using a Split-Process-Merge flow graph](image)
The Split-Process-Merge construct shows all the basic building blocks for parallel schedules, and is suitable for simple compute farm applications. We can distinguish three basic operation types in this flow graph:

- **Split** operations – these operations take exactly one data object as input, and can produce any number of data objects as output (but at least one, otherwise the data flow in the flow graph would be halted). Outputs typically represent subtasks that may be performed in parallel.
- Processing operations (also called Leaf operations) – these operations take exactly one data object as input, and produce exactly one data object as output. They typically process requests sent by the preceding split operation.
- **Merge** operations – these operations take any number of data objects as input, and produce exactly one data object as output. The number of input data objects corresponds to the number of data objects sent by the corresponding split operation. These operations are used to merge the partial results produced by the leaf operations into a single result.

The actual behavior of each operation is specified by the application developer. In particular, the split and merge operations contain customized code to control exactly how data is distributed in subtasks, and how the computed sub-results are combined into the final output result. The data objects that drive the execution of the flow graph can also be freely defined by the developer. The mechanism used for allocating processing nodes for the operations is described later.

### Complex flow graphs

Operations can be combined to create flow graphs of arbitrary complexity, provided that every split is matched by a single merge operation. Three fundamental patterns can be distinguished in flow graph construction: pipelines, nested constructs and parallel branches. Additional constructs, such as stream operations and loops, can be used in order to construct complex pipelined flow graphs. **Pipelines** are the simplest pattern, where two leaf operations follow each other. From the outside perspective, a sequence of two leaf operations is equivalent to a single leaf operation, since for each input data object, there is exactly one output data object. A basic pipelined sequence of leaf operations is shown in Figure 3. This equivalence property also applies to the simple split-process-merge flow graph, since for every data object that is input into the flow graph, exactly one data object is output, as illustrated in Figure 4. The leaf operation within the split-process-merge sequence can thus be replaced by another split-process-merge graph, yielding a nested graph, as illustrated in Figure 5.

![Figure 3](image1.png)

**Figure 3.** (a) Pipelined sequence of operations and (b) equivalent outside view of a pipelined sequence of operations

![Figure 4](image2.png)

**Figure 4.** (a) Split-Process-Merge flow graph and (b) equivalent outside view of split-process-merge flow graph section
Different outputs of a split operation can also be processed by different operations, yielding multiple parallel branches on the flow graph between the split operation and the merge operation.

**Stream operations**

The split-merge concept used in parallel schedules provides a very strict synchronization model. Applications often need to perform multiple computation steps, where the following computations need the results of previous computations in order to execute. When these computations are split up and performed in parallel, the synchronization usually is performed by ensuring that all the first computations have completed with a merge operation before splitting out the second computation as illustrated in Figure 7.

In some cases, not all the results of the parts of the first computation are required in order to perform parts of the second computation. In these cases, it is desirable to already split out these parts of the second computation before the first computation is complete in order to ensure proper pipelining of the application. The stream operation was designed to accomplish this objective, by combining the functionality of a merge operation and a split operation in a single operation, as illustrated in Figure 8. The stream operation can output data objects at any time within the merge process.

Stream operations can therefore be used for improving pipelining across synchronization points. An example illustrating the use of the stream operation and the improved pipelining that can be achieved is provided in the Advanced Tutorials chapter, within the context of an LU decomposition application.
Loops

Some applications require the repeated execution of a sequence of operations until a certain condition is met. Within a flow graph, this type of execution pattern can be expressed using a loop, a specialized type of operation that encloses any sequence of operations and evaluates a condition on the output data object of this operation. As long as the condition is true, the encapsulated sequence of operations is executed again on the output data object. The encapsulated sequence of operations needs to have the same input and output data object types in order to ensure that looped execution is valid. The loop operation does not create a cycle within the flow graph; it produces a pipelined sequence of operations, the length of which is determined at runtime based on the loop condition.

Figure 9. (a) Loop construct enclosing a leaf operation and (b) equivalent pipelined sequence of operations if the condition returned true on the first three iterations

Figure 10 illustrates some additional flow graph constructions that can be achieved using loop constructs. When inserting loop constructs, the only constraint is to ensure that the flow graph preserves the symmetry between split and merge operations: for each split operation within the flow graph, a corresponding merge operation needs to exist within the flow graph.

Figure 10. Other configurations using the loop construct (left) and resulting runtime flow graph (right). (a) multi-level split and merge operations, (b) deep pipeline using stream operations

By combining the operations and constructs shown in the previous sections, flow graphs of arbitrary complexity may be created. The flow graphs resulting from the basic patterns are always acyclic, since none of the patterns allows the construction of cycles. The acyclic property is particularly important for program execution, since it ensures that execution is always free of deadlocks.

Threads and thread collections

Flow graphs provide a simple and efficient mechanism for describing program flow. We now need to indicate on which processing node the various operations on the flow graph should be executed. The processing node selection is performed by assigning operations to threads that are mapped onto the processing nodes. A thread within the parallel schedules concept provides a context within which operations can be executed. A parallel schedule thread typically executes only one operation at a time. However, multiple parallel schedule
threads can be located on the same processing node, and execute operations concurrently independently of each other. The threads are grouped within thread collections, which contain multiple threads of the same type. The threads within a thread collection may be located on an arbitrary number of processing nodes. Let us for instance consider a simple compute farm application, where a master node distributes tasks to a set of processing nodes, and later collects the results of the processing. For such an application, two thread collections would be created, where the first is used for all the master tasks, and the second is used for all the processing tasks. The master thread collection contains only a single thread, whereas the processing thread collection contains one thread for each participating processing node.

![Diagram](https://example.com/diagram.png)

Figure 11. Example of thread collections for a simple compute farm application. Two thread collections, Master threads and Processing threads, are used.

In order to use the thread collections, they need to be attached to the operations within the flow graph. For each operation within the flow graph, one thread collection is chosen. In our compute farm example, the Split and Merge operations are attached to the Master threads collection, and the ProcessData operation is attached to the Processing threads collection.

![Diagram](https://example.com/diagram.png)

Figure 12. Assignment of thread collections to operations within the flow graph

**Routing functions**

This leaves one question unanswered during program execution: which thread within the collection should be used as the execution context for running an operation? To answer this question, an additional function is added to all operations within the flow graph: the routing function. The role of the routing function is to select a thread within the assigned thread collection. Since execution of operations is triggered by data objects in our data-driven computation model, the routing function can use the data object that will be processed as input by the operation as a parameter. The routing function is very flexible, since it can provide both simple mechanisms such as constant routing or round-robin routing, and complex mechanisms such as data-dependent routing (by using the data object) or even automatic load-balancing. For the simple compute farm example, the routing functions shown in Figure 13 can be used.

![Diagram](https://example.com/diagram.png)

Figure 13. Routing functions for compute farm

The threads and thread collections provide a logical description of the execution environment of the application, since at this point the threads have not yet been assigned to processing nodes in the cluster. In order to complete
the description, each thread of each collection must be mapped to the processing node where its assigned operations are to be executed. Multiple threads (whether from the same or from different thread collections) may be mapped to the same processing node. This assignment process is known as thread mapping.

**Thread local storage**

Threads, parallel schedule threads can provide local storage to the operations that execute within their context. The storage is provided as an instance of a user-defined data structure. This local storage is preserved within the thread state, and persists from one operation to the next. Data-parallel applications can use this thread local storage to store distributed data structures. For example, an application performing matrix computations could store matrix blocks within its threads. The storage is specific to individual threads. Even when two threads of the same type are located on the same processing node, they cannot access each other's data.

**Runtime behavior of parallel schedules**

All communications between DPS operations are performed asynchronously. Each thread therefore stores incoming data objects within a queue until it can process them. For instance, let us consider the execution of a simple split-process-merge compute farm parallel schedule in a situation where the split operation creates 6 subtask data objects. Figure 14 illustrates the fully unfolded parallel schedule view, showing all the operations that are executed during program execution and all the data objects. Every oriented edge in the unfolded view of the parallel schedule represents a data object, ending in the operation that processes the data object. Since the fully unfolded view can easily become overcrowded if the application creates large numbers of data objects, and therefore executes many operations, we often use a partially unfolded view showing only a single instance of each operation for every thread. On the partially unfolded view in Figure 14, the data objects are also shown, illustrating that each operation is executed twice on each thread.

Since a thread can only execute one operation at a time, the partially unfolded view is actually a representation of the runtime behavior of the parallel schedule. For instance, when the execution of the parallel schedule is started by the initial request data object, the split operation starts execution, and produces 6 output data objects. These output data objects are distributed onto the 3 processing threads by using the round-robin routing function, each thread receiving two data objects. The incoming data objects are stored within the thread’s data object queue. As soon as a data object is available within a thread’s data object queue, that thread immediately starts processing by executing the appropriate operation. When the execution of the operation completes, the next data object within the queue is retrieved for processing. The arrival of the output data objects from the split operation will thus cause the three processing threads to start execution in parallel, and to send their output result back to the master thread, which initiates the merge operation. The timing diagram for the complete program flow is shown in Figure 15.

![Diagram showing runtime behavior of parallel schedules](image-url)
Data objects are sent to the target thread as soon as they are created by an operation. This behavior, combined with the use of queues to store incoming data objects, allows the overlapping of computations and communications and (at least partially) hides the latencies and transfer times related to network communications.

**Flow control**

Since the parallel schedule execution model sends data objects as soon as they are created, a split operation that creates many output data objects induces a high network load. Moreover, the storage of these data objects may consume large amounts of memory within the target threads. In order to resolve this problem, parallel schedules provide a flow control mechanism that limits the number of data objects that can be in circulation at a given moment. The limitation is achieved by introducing a feedback loop between a split-merge operation pair. The split operation is initially permitted to emit a fixed number of data objects, after which it is suspended. For each data object that is received by the corresponding merge operation, the split operation may emit a new data object.

**Load balancing**

When using simple compute farm patterns with a routing function such as round-robin routing, load imbalance appears when the distributed tasks are not all of equal complexity or when the individual threads do not have the same processing power available to each of them. The distribution of tasks is controlled by the routing function attached to the first operation after the split operation in the flow graph. In order to achieve load balancing, this routing function has to select threads based on their effective computation throughput, thus sending more data objects to the threads that can process more operations.

DPS provides such a routing function by taking advantage of the feedback loop introduced for the flow control mechanism. Instead of maintaining a constant number of data objects in circulation between the split operation and merge operation on a global scale, the number of circulating data objects is maintained constant for each thread that can be reached from the split operation. This is achieved by storing the index of the path used for routing within the data objects sent by the split operation, and preserving this value until the data object arrives.
at the merge operation. The merge operation then returns this index to the split operation in its flow control notification. The next data object created by the split operation will be sent along this path, thus maintaining the number of data objects on each path constant. In order to initialize the system, the routing function initially routes the data objects according to a round-robin scheme up to the limit imposed by flow control.

![Flow graph](image)

Figure 17. Load balancing in parallel schedules. When the split operation sends out a new data object, it is sent to the thread that last returned a data object to the corresponding merge operation.

The load balancing mechanism can effectively equilibrate uneven computation loads when the number of data objects representing subtasks is significantly greater than the number of threads used for computation.

**Flow graph construction**

All of the elements that are used to build flow graphs such as operations or routing functions are implemented within the application's source code as C++ classes. These statically defined components are assembled at runtime to build the final flow graph. The flow graph can subsequently be used to execute parallel schedules. Flow graphs can be assembled at any time during execution, and can be used to spawn any number of parallel schedules. Flow graphs can contain any combination of operations, as long as the fundamental symmetry of split and merge operations is preserved. The syntax of these elements is presented in the Reference chapter.

**Connecting operations**

Operations constitute the nodes of the flow graph, and encapsulate all of the application's functionality. Within DPS, all operation types (*leaf*, *split*, *merge* and *stream*) share a common syntax and programming model. The body of an operation is composed of standard sequential C++ code that performs the operation's tasks. Operations are further characterized by three external parameters: the acceptable input data object types, the type of output data object produced, and the type of thread used for local data storage. When a flow graph is built, the sequence of operations is validated at compile time to ensure that the input types of an operation match at least one output type of the preceding operation.

**Attaching thread collections**

In order to have an execution context, operations need to be attached to thread collections. As part of this attachment, a routing function is also assigned to the operation. As with the interconnection of operations, DPS performs type validation in order to ensure that the operations will be executed on threads of the appropriate type, and that the routing functions use the correct data object type in order to select the target threads. Matches are not strict however, as many routing functions are not data dependent, and therefore do not care about the data object type that is to be routed. Similarly, operations that do not perform any processing on a locally stored thread state can be attached to a thread collection containing threads of any type. For these cases, any data object type or thread type will pass the validation.

**Conclusion**

This closes the overview of the concepts underlying the Dynamic Parallel Schedules framework. These are the basic concepts used by the applications described in the Basic Tutorials chapter. Additional advanced features, such as fault-tolerance, malleability, parallel components and performance prediction are described in the Advanced Tutorials, Simulation & Performance Prediction and Message Race Detection chapters.
**Getting Started**

**Installing DPS**

The DPS library and the applications described in this tutorial are available as archives of source files on the dps.epfl.ch website. The library and the applications are built by executing `make` in the directory where you have extracted DPS on UNIX systems, or by building the `dps.sln` Visual Studio solution on Windows systems. The following paragraphs provide more detailed build instructions and compatibility information in case the simple technique does not work.

**Building on UNIX systems**

DPS provides a set of makefiles for building the static DPS library and the sample applications on UNIX systems. The build process requires GNU make and a `pthreads` library, and a compiler that supports partial template specialization, such as `gcc` version 3.0 or newer. The networking code requires either the MPI library or raw TCP sockets.

DPS has been tested successfully on Linux (x64 and x86), Solaris, Mac OS X (Intel and PPC) and Cygwin using the gcc compiler. For other systems, you may have to modify the makefiles or the source code to compensate for different header files, type definitions, and libraries.

**Library**

After unpacking the DPS library, go into the root directory and type `make`. The default build requires a multithreaded MPI2 library and assumes that `mpicxx` is available. This builds the DPS library in `dpsnext/libdps.a`. The following options can be used to customize the build process:

```
make [TCP=true][DEBUG=true][SIM=true]
```

The `TCP` define compiles the library and applications using TCP networking instead of MPI. The `DEBUG` define will add debug information to the library. The `SIM` define will enable the application simulation and testing capabilities described in the chapters Simulation & Performance Prediction and Message Race Detection. The `TCP` and `SIM` defines are mutually exclusive.

**Tutorials**

The makefile of each tutorial program assumes that the library can be found in `../dpsnext`. You must therefore either copy the library directory into the tutorials directory, or update the `DPSDIR` variable in the makefiles to ensure that the compiler can find the required header and library files.

Each program is built using `make`. Building for the TCP network layer or for simulations requires setting the `CXXFLAGS` variable to `–DDPS_TCP` or `–DDPS_SIM` respectively. This can be done either in the makefile itself or on the command line:

```
make [CXXFLAGS=–DDPS_TCP][CXXFLAGS=–DDPS_MPI]
```

Add (or set) `–g` to `CXXFLAGS` to compile with debug symbols.

Most of the programs in the advanced_tutorials archive require features unavailable with MPI, and build by default using the TCP networking layer.

**Building on Windows systems**

It is possible to build DPS using the free Cygwin environment; in this case the UNIX build instructions above apply.

The ZIP archives on the dps.epfl.ch website include project files for Visual Studio 2005. The compiler of Visual Studio 2003 is also supported. Since it requires partial template specialization, DPS will not compile with earlier Microsoft C++ compilers. The default configuration uses MPI for all network communications, and requires an MPI2 implementation with support for multithreading (MPI_THREAD_MULTIPLE). If MPI is not available, an implementation using raw TCP sockets is also provided.
After unpacking the distribution, open the provided DPS project file (dpsnext.sln) and build the library. The build process puts the DPS library in the bin directory. The solution includes Release and Debug configurations for both MPI and TCP. If you use MPI, you may have to update the include and library directories in order to compile the library. The Sim Debug and Sim Release configurations enable the application simulation and application capabilities described in the chapters Simulation & Performance Prediction and Message Race Detection.

The basic_tutorials archive includes the DPS kernel and a few sample programs that are compiled in the bin directory. You must either copy the library directory into the tutorials directory, or update the projects with the location of the library such that the compiler can find the required header and library files.

Deploying parallel applications using MPI

DPS applications that use the MPI network layer are started like any MPI application, e.g. using the mpiexec or mpirun launchers. Within the application, the MPI processes are identified using the process rank in the MPI_COMM_WORLD communicator.

Deploying parallel applications using TCP

If you use the TCP networking layer, DPS supports the dynamic allocation and deallocation of compute nodes at runtime. These are performed through connectors, which controls how new instances of applications are started on the participating nodes. DPS provides three basic connectors: RSHConnector, KernelConnector, and LocalConnector. The connectors are responsible for the addressing mechanism used for identifying nodes and executing new instances of the parallel application.

The RSHConnector is designed to use existing operating system services for starting remote applications, such as rsh or ssh. These remote shell utilities are commonly installed on clusters running Unix-based operating systems. However, startup times for remote shell sessions, in particular with ssh, can be fairly high, leading to long application startup times.

The KernelConnector uses a DPS-specific application launcher, called a kernel, which needs to be running on all nodes that are participating in the execution of the DPS application. The kernel provides faster startup times, and additional features such as resource monitoring.

Both connectors require the executable and data files to be available on the remote host. The DPS kernel in particular does not provide any facilities for sharing or copying executables and data files. The easiest solution to making the files accessible is to use some form of shared directories, as provided by NFS or Windows network shares. It is also possible to replicate the files to individual nodes. The executables do not need to be in the same location on all nodes, as DPS provides a mechanism for dynamically renaming executables. However, it is the responsibility of the application developer to ensure that remote application instances can find the required data files.

The LocalConnector starts multiple instances of the same application on the local host, and is therefore not subject to file replication issues. It is mainly used for debugging and testing purposes, or to run an application on a multiprocessor machine. The full details are available in the Reference chapter.

Networking

When using the TCP network protocol, individual instances of DPS applications are identified by the IP address of the host they are running on and the port number on which they are listening for incoming connections. When multiple instances of DPS applications are running on the same host, they must use distinct port numbers. The mechanism used for assigning the port number is dependent on the connector being used and the DPS library configuration.

When using the rsh connector, the port number must be specified on the command line or in a configuration file. When additional instances of an application are started on remote processing nodes, they listen on the same port number as the initial application instance. The user needs to ensure that no collisions in port numbering occur when multiple applications are sharing the same node.

When using the kernel connector, the port number allocation is handled by the kernels. All kernels listen on a fixed port specified on the command line or a configuration file. Port numbers are dynamically assigned to the applications as they are executed by the TCP/IP implementation, and the kernel keeps track of the running applications in order to ensure that they can be contacted by remote instances. Therefore using the kernel is ideal when multiple DPS applications are sharing the same cluster, since port number conflicts are avoided even when running multiple instances of the same application on the same nodes.

When using the local connector, multiple instances of an application are identified only by their port number, since they are all running on the same node. The port numbers are read by the local connector from the thread mapping string, and the new instances are launched with the appropriate port.
Heterogeneous execution environments

The DPS library supports the execution of programs on heterogeneous clusters. The network protocols used by the DPS library for transferring data objects are transparent to platform differences such as data endianness and structure alignment. The application (and the kernel if the kernel connector is used) must be compiled for each one of the different architectures and platforms that will be involved in the computation. The application is then executed as if it was on a homogeneous cluster.

This feature has been successfully tested with a mix of 32-bit and 64-bit x86 Linux and a PowerPC Macintosh, as well as on a Solaris SPARC and Windows deployment.
Basic Tutorials

Introduction

These tutorials provide a step-by-step introduction to the features of the DPS framework. The first tutorial provides a quick tour of how to build a simple parallel application. The second tutorial extends this application to implement pipelined execution, conditional execution, and some more advanced DPS concepts. The next two tutorials present two applications that use a real workload: a parallel computation of the Mandelbrot set, and a parallel version of Conway's game of life. The game of life exhibits communication patterns that are commonly found in many traditional parallel applications. The full source code to all the applications used in the tutorials is provided in the DPS distribution, together with the makefiles required to build them.

A simple DPS application (uppercase)

This first tutorial section will guide you step by step through the creation of a simple DPS application. It presents the DPS classes that are used to define the various components of a parallel schedule. The full source code for this tutorial can be found in the uppercase sample application.

Introduction

Let us first write a simple parallel application that converts a text string from lowercase to uppercase characters. The parallelization is obtained by splitting the string into individual characters, and by converting the characters on multiple machines. This can be implemented using a basic Split-Leaf-Merge flow graph.

- The program starts with a data object containing a complete text string that is segmented into single characters by the SplitString operation.
- The ToUpperCase operation then receives a single character from SplitString, and outputs another character that is the upper case form of the input.
- The MergeString operation then collects all single upper case characters and puts them back into a single string.

Remember that DPS flow graphs must be symmetric: for each split operation, there must always be a corresponding merge operation. In addition, a merge operation shall receive as many data objects as sent by the matching split operation.

Figure 18. Flow graph for the ToUpperCase application

Header files and namespaces

All DPS applications need to include the DPS header file dps.h. This header file provides the declarations for all DPS classes. All DPS classes and functions are in the dps namespace. This can either be included with a using namespace dps directive, or all DPS classes can be prefixed with the dps:: namespace specifier.

```cpp
#include <dps/dps.h>
```

Data object definitions

Our application needs two types of data objects, representing respectively a string of characters and a single character. The following source code is used for the declarations:

```cpp
#define STRLEN 26

//! A data object containing a single fixed-length string
class StringData : public dps::SimpleSerial
```
The `StringData` object contains a simple fixed-length array of characters representing the string to process. The `CharData` object contains a single character and the position of the character within the string. The position is required in order to be able to place the character at the correct position within the output string, since the characters might be received out of order in `MergeString`.

In order for objects to be sent and received over the network, they must be serializable. Since C++ does not provide any mechanisms to handle serialization and deserialization automatically, DPS provides one. In this example, both data objects derive from the base class `dps::SimpleSerial`, which provides a mechanism that transfers data by performing a simple memory copy.

When a data object is received as a stream of bytes on a node, the corresponding object must be instantiated and initialized. DPS provides a class factory that can instantiate classes that were previously registered. The registration is performed by the `IDENTIFY` macro, which will be found in most user-defined classes.

The serialization model provided by `dps::SimpleSerial` is the simplest to use, but it obviously has many shortcomings: the data must be of constant size, and may not contain any complex data elements such as pointers or STL containers. Since the serialization is dependent on the placement in memory of the structure members, `dps::SimpleSerial` cannot be used in heterogeneous environments. Another base class is provided for automatically serializing complex data types and will be presented in a later tutorial.

**Definition of ToUpperCase operation**

DPS uses C++ classes to represent operations. All operation types share a common syntax and programming model, and must derive from a base class provided by the DPS library. The base class is a template, taking as arguments the input data object type and the output data object type. (These are used to validate the construction of the flow graph.)

The first operation we will define is the `ToUpperCase` operation, which transforms a single character from lower case to upper case. This operation is defined as follows:

```cpp
// ToUpperCase operation
// The operation takes a CharData data object as input and output
class ToUpperCase : public dps::LeafOperation<CharData, CharData>
{
    IDENTIFY(ToUpperCase);
    public:
        void execute(CharData *in)  // Pointer to input data object
        {
            // Post equivalent uppercase character
            postDataObject(new CharData(toupper(in->chr), in->pos));
        }
};
```
The ToUpperCase operation derives from \textit{dps::LeafOperation}, which provides the basic functionality for simple operations. Since both the input and output data objects are of type \textit{CharData}, the two template parameters are identical.

As for all other DPS classes, there is an \textit{IDENTIFY} macro in order to allow the framework to instantiate the operation when it needs to be executed.

An operation creates its output data objects by allocating them with the standard C++ \texttt{new} operator, and then transfers them to the next operation on the graph by calling the \texttt{postDataObject} function. Within \texttt{postDataObject}, the DPS library performs all required routing, serialization, etc.

In the ToUpperCase operation we simply create a new \textit{CharData} containing the uppercase equivalent of the character in the input data object.

\textbf{Data object lifetime}

DPS automatically takes care of the destruction of the data objects passed to its functions, and also destroys the input data objects of operations after their execution. Here, the \textit{in} data object is destroyed at the end of the operation, while the posted \textit{CharData} object is destroyed once sent over the network. The user does therefore not need to delete them explicitly.

There are some cases where we do not want an object to be destroyed, e.g. when we send a same data object multiple times, or when we want to store a data object so that it can be used by another operation. DPS uses a reference counting mechanism for managing the data objects' lifetime. The reference count can be modified by calling the \texttt{addRef} (increment reference count) and \texttt{release} (decrement reference count). When a data object is created, its reference count is one. The data object is deleted when its reference count reaches zero. Here is a variant of the ToUpperCase operation that does not create a new data object:

\begin{verbatim}
void execute(CharData *in)  // Input data object, reference count is 1  
{  
in->addRef();  // Reference count is 2  
in->chr=toupper(in->chr);  // Modify input data object  
postDataObject(in);  // Calls in->release(), reference count becomes 1  
}  // in is released once again when the operation terminate  
// Reference count drops to 0 and the data object is deleted
\end{verbatim}

\textbf{SplitString and MergeString operations}

The SplitString operation is responsible for cutting the input character string into individual characters. The SplitString operation derives from \textit{dps::SplitOperation}, which provides the basic functionality for split operations. This operation receives a \textit{StringData} object as input and generates \textit{CharData} objects as output. It creates multiple output data objects in a simple C++ \texttt{for} loop and posts them.

\begin{verbatim}
// Split operation  
class SplitString : public dps::SplitOperation<StringData,CharData>  
{  
IDENTIFY(SplitString);  
public:  
void execute(StringData *in)  
{  
    for(int i=0;i<STRLEN;i++)  
        postDataObject(new CharData(in->str[i],i));  
    }  
};
\end{verbatim}

The MergeString operation derives from \textit{dps::MergeOperation}, which provides the basic functionality for merge operations. The operation is called when the first data object arrives, and then creates the output data object. It then waits for the remaining data objects corresponding to those created by the split function by calling the DPS library function \texttt{waitForNextDataObject}. The character contained in each input data object is copied to the appropriate position in the output data object. \texttt{waitForNextDataObject} returns a \texttt{NULL} pointer when all data objects have been received. The output data object is posted when all data objects have been received.

\begin{verbatim}
// Merge operation  
class MergeString : public dps::MergeOperation<CharData,StringData>  
{  
    void execute(CharData *in)  
    {  
        postDataObject(in);  
    }  
};
\end{verbatim}
Routing functions

We now need routing functions to compute the destination thread of each data object. The function returns the index within the thread collection of the thread to use. It can use any fields of the data object being routed, as well as the size of the destination thread collection in its computations.

The first route, MainRoute, is used for routing data objects to the main thread. Since the thread collection contains only a single thread, the routing function returns a constant index of 0. In the general case, the thread collection on which the destination operation is mapped may contain any number of threads. The number of threads can be retrieved with the threadCount member function. The MyRoundRobinRoute uses this function to route the characters to successive processing threads according to their position in the string. The index is computed in the present example as the position of the character in the string modulo the number of available threads.

```
// Routing functions
class MyRoundRobinRoute : public dps::Route<CharData>
{
    IDENTIFY(MyRoundRobinRoute);
    virtual Size route(CharData *in) { return in->pos%threadCount(); }
};

class MainRoute : public dps::Route<dps::ISerializable>
{
    IDENTIFY(MainRoute);
    virtual Size route(dps::ISerializable *in) { return 0; }
};
```

The template parameter indicates the data object type used for deciding the route. On the MainRoute above, this parameter is dps::ISerializable, which is the base class for all DPS serialization models, and used when the data object type is not important. The route method returns the index of the thread to which the data object is to be routed. The actual thread collection that is used as the target for the routing is specified in the definition of the flow graph, which will be described later. The index can be computed as a function of any members of the current data object, as well as of the number of threads in the thread collection.
Since these routing scenarios are very common, DPS provides built-in routing functions that have the same functionality: `dps::ZeroRoute` and `dps::RoundRobinRoute`. These routing functions (and others) will be used in later tutorials.

**Application class**

The application class derives from `dps::Application`, and contains the startup function of the application, `start`. (See chapter Reference for a description of the other member functions, `help` and `init`.) This function is called after the DPS library has been initialized. The usual `IDENTIFY` macro is also present here.

```cpp
// Application
class ToUpperCaseApp : public dps::Application
{
public:
    // Application startup function
    virtual void start();

    IDENTIFY(ToUpperCaseApp);
};
```

The startup function contains code to create the thread collections and flow graph, and is also responsible for running the parallel schedule. The following paragraphs present in detail the code contained in the body of the startup function.

**Creation of the thread collections**

Thread collections contain the logical threads that will be used for executing the parallel application. These threads are mapped onto operating system threads distributed over one or more nodes of the cluster. For this application, two stateless thread collections are created, called `main` and `process`. The thread collections do not contain any threads initially. The threads are added in a later step, and can be modified at any time.

```cpp
dps::StatelessThreadCollection theMainThread =
    getController()->createStatelessThreadCollection("main");
dps::StatelessThreadCollection processThreads =
    getController()->createStatelessThreadCollection("process");
```

A stateless thread collection contains threads that do not store any local data. Threads with local data will be presented further on. The names of the thread collections can be used to retrieve them at other points in the application. The DPS controller object obtained with `getController` is the entry point to most DPS functions. After creating the thread collections, we add an initial set of threads to the collections with the `addThread` function. When threads are added to the collections, DPS will automatically execute the application on the remote nodes as required. The format of the addresses used in the calls to `addThread` depends on the underlying network layer being used.

If you use the default MPI network layer, processes are addressed using their rank in the default MPI communicator, i.e. integers between 0 and $n-1$, where $n$ is the number of MPI processes. If you use standard TCP/IP sockets for communications, addresses are `hostname:port` or `IP:port` strings; the hostname specifies where the remote application must be launched, and the port number indicates where the remote instance will listen for incoming requests. This tutorial uses MPI addresses by default. However, the Reference section provides more details about the use of raw TCP sockets.

```cpp
Result r = theMainThread.addThread("0");
if(DPS_FAILED(r))
{
    upperLog.write(0) << "Could not map main thread collection";
    dps::FatalError::stop();
}
r = processThreads.addThread("0 1");
if(DPS_FAILED(r))
{
    upperLog.write(0) << "Could not map process thread collection";
    dps::FatalError::stop();
}
```
The above example creates one thread for the *main* thread collection, and two threads in the *process* thread collection. The thread of *main* and the first thread of *process* will run within the same process (address space). The other thread of *process* will reside in the process with rank 1. For debugging purposes, it is often interesting to have all threads within the same process in order to enable the use of a conventional debugger. This can easily be achieved by specifying the same host multiple times in the mapping. The following example creates three threads within the same address space.

```cpp
if (DPS_FAILED(processThreads.addThread("0 0 0")))
{
    upperLog.write(0) << "Could not map process thread collection";
    dps::FatalError::stop();
}
```

### Relationship of DPS threads to operating system threads

When threads are added to a thread collection, DPS typically creates one operating system thread for each DPS thread. There are two mechanisms in DPS that can be used to control this creation process: *aggregation* and *simultaneous execution*. Simultaneous execution can be used to map multiple operating system threads to a single DPS thread in order to allow multiple operations to overlap within the execution context provided by the DPS thread. Simultaneous execution is presented in more detail in a later tutorial, 'Neighborhood-dependent parallel applications'.

Aggregation is used when multiple DPS threads are created within the same application instance, for example by using a mapping string such as "0 0" in the example above. In this situation, having two operating system threads on the same node is not efficient if the node has a single processor. Therefore, DPS may share one operating system thread for both DPS threads, ensuring that only one of the DPS threads is executing an operation at any time. This sharing process is called *aggregation*.

Both options can be specified in the construction of the thread collection as follows:

```cpp
dps::StatelessThreadCollection processThreads =
    getController()->createStatelessThreadCollection("process",1,true);
```

The first argument of `createStatelessThreadCollection` specifies the name of the thread collection. The second argument is the number of simultaneous execution threads desired, and the third argument specifies whether aggregation should be used. The default number of simultaneous execution threads is 1, and aggregation is disabled by default.

### Definition of the flow graph

The flow graph indicates the program flow, the routing of data objects and the mapping of operations to thread collections. The flow graph is constructed in two parts. Initially the various segments that compose the graph are created by combining a sequence of `dps::FlowgraphNode`. These sequences are collected within a `dps::FlowgraphBuilder` object. Once all the sequences have been added to the `dps::FlowgraphBuilder`, the complete graph is instantiated with a call to `createFlowgraph` in the controller object.

```cpp
dps::FlowgraphBuilder theGraphB =
    dps::FlowgraphNode<SplitString,MainRoute>(theMainThread) >>
    dps::FlowgraphNode<ToUpperCase,RoundRobinRoute>(processThreads) >>
    dps::FlowgraphNode<MergeString,MainRoute>(theMainThread);
```

The first template argument for the `dps::FlowgraphNode` is the operation that is to be executed. The second template argument is the routing function that is used to route the input data object of the operation to the appropriate thread. The thread collection within which the operation will be executed is the parameter of the `dps::FlowgraphNode` constructor. The flow graph nodes are combined by the `>>` (right shift) operator that has been overloaded within the DPS library. For the simple flow graph we are using here, a single sequence of nodes is used and directly assigned to a `dps::FlowgraphBuilder`. The subsequent call to `createFlowgraph` takes two arguments: the name of the flow graph, and the flow graph builder representing the flow graph. Figure 20 shows the resulting flow graph, partially unfolded on two threads.
The first flow graph node executes the operation \textit{SplitString}, and all incoming tokens are routed with \textit{MainRoute}, to the single thread of the \textit{theMainThread} thread collection.

The second flow graph node executes \textit{ToUpperCase}, and incoming data objects are routed with \textit{RoundRobinRoute}. Suppose there are two threads in the \textit{processThreads} thread collection. All characters with an even position are sent to the first \textit{ProcessThread}, all odd characters to the second.

The last flow graph node executes \textit{MergeString}, and all incoming data objects are routed with \textit{MainRoute}.

Figure 20. Flow graph for the ToUpperCase application

**Execution of the parallel application**

The last part of the \textit{start} function launches the parallel application. It first creates the input data object of the parallel schedule. The parallel schedule is then invoked by calling the \textit{callSchedule} method of the DPS controller. This method is a blocking call, and returns the output data object of the schedule. The returned output data must be released explicitly when it is no longer needed.

```cpp
StringData *in, *out;
in = new StringData();
std::cout << "Input string: " << in->str << std::endl;
out=(StringData *)getController()->callSchedule(theGraph,in);
std::cout << "Output string: " << out->str << std::endl;
out->release();
```

**main function**

Finally, as for any C++ application, the DPS application needs a main function, where program execution starts. For this simple application, we simply start the DPS library by calling \textit{dps::dpsMain}.

```cpp
int main(int argc, char *argv[])  
{  
    return dps::dpsMain(argc, argv, new ToUpperCaseApp());  
}
```

The arguments to the \textit{dps::dpsMain} function are the command line parameters and a pointer to the DPS application to be executed. In this example we create a new instance of our application class \textit{ToUpperCaseApp}. The \textit{main} function will obviously be executed on all nodes used by the application, since it is the entry point for the program. The \textit{dps::dpsMain} function will initialize the DPS library, and call the \textit{start} method of the application on the first node of the application.

**Advanced DPS constructs (uppercasex)**

This second tutorial section presents extensions of the uppercase sample application. It presents thread local variables, conditional program flow, and operation pipelines. The complete source code for all the flow graphs described here can be found within the \textit{uppercasex} sample application. The \textit{uppercasex} application executes the four presented graph variants successively, and displays their output strings.
Local variables in threads

The first part of the tutorial used a set of stateless threads for running the ToUpperCase application. The threads can also be used to store local data that can be used when executing operations. The threads are expressed as standard C++ classes. The thread instance in which an operation is running can be accessed by calling its getThread member function. The following class is a simple thread we will use in our application. For simple applications, the thread classes need not derive from any particular base class, since they will not be moved from one node to another. (More advanced DPS features such as malleability and fault tolerance described in the next chapters require that threads use a serialization model.)

```cpp
// ProcessThread logical thread
class ProcessThread
{
  IDENTIFY(ProcessThread);
public:
  UInt32 passNumber;

  // Constructor to initialize thread local variable.
  ProcessThread() { passNumber=0; }
};
```

The operations that will be running on the thread need to specify the thread type within the template arguments of their operation base class. In the present example, we will use the ProcessThread thread type for the process thread collection. Within the ToUpperCase operation, it is then possible to use getThread()->passNumber to access the passNumber variable of the thread. Unlike operations, which are destroyed after execution, threads are created once. They therefore preserve their variables, which can be used by multiple operations that run on the same thread. If you wish to initialize such variables, a constructor without any parameters can be added to the thread. This constructor is called when the thread is created by DPS.

The following source code shows the modified ToUpperCase operation that uses the pass number in order to convert only a limited number of characters to uppercase. The thread type used for the operation is specified as the third template parameter to the operation base class. The first and second template parameters are respectively the input and output data object types.

```cpp
class ToUpperCase : public dps::LeafOperation<CharData,CharData,ProcessThread>
{
  IDENTIFY(ToUpperCase);
public:
  void execute(CharData *in)
  {
    if(getThread()->passNumber<2)
      postDataObject(new CharData(toupper(in->chr),in->pos));
    else
      postDataObject(new CharData(in->chr,in->pos));
    getThread()->passNumber++;
  }
};
```

When the parallel schedule with the modified ToUpperCase operation is run with three threads in the process thread collection, only the 6 first characters of the output string are converted to uppercase.

Conditional dataflow

In some cases we want different outputs of a split operation to be processed by different type of operations. Since data object are routed along the flow graph according to their type, a conditional dataflow can be created by emitting data objects of different types. For example, in order to perform the ToUpperCase operations on the first 8 characters of the string independently of any thread local state, a second data object type CharData2 identical to CharData (except for the class name) can be used. The CharData2 type could be defined as follows.

```cpp
// For conditional execution, another name for CharData
class CharData2 : public CharData
{
  IDENTIFY(CharData2);
};
```
The class `CharData2` simply derives from `CharData`, inheriting all the members of `CharData` and allowing it to be used anywhere a `CharData` is appropriate. Since the class contains no additional data compared to `CharData`, there is no requirement to add any specific support for serialization (the members inherited from `CharData` will do the work). The `IDENTIFY` macro is nonetheless required to ensure that the class is registered in the class factory separately from `CharData`. The split operation is modified to output 8 data object of type `CharData`, followed by 18 objects of type `CharData2`.

Since the `SplitString` operation can now generate two data object types, both of these need to be mentioned in the output data types parameter of the operation template. The two data object types are specified with the helper template `dps::tv` (short for type vector), which can be used to build lists of types. The `dps::tv` template can take from one to five arguments, indicating the valid types. Flow graph nodes cannot be attached together in the flow graph if the output data objects of a node are not valid input types for the next node in the flow graph. The `ToUpperCase` operation is modified to output only `CharData2`, and the merge operation is therefore modified to accept only `CharData2`.
Finally, we need to create a flow graph that contains two paths: one for the characters that need to transit through `toUpperCase`, and another for those that bypass it, as shown in Figure 21.

```
while((in=waitForNextDataObject())!=NULL);  
postDataObject(out);
```

![Flow graph with type-conditional routing](image)

**Figure 21.** Flow graph with type-conditional routing

Multiple branches in the flow graph are added one after another to the flow graph builder object. The common flow graph nodes are indicated by using the same object instances of `dps::FlowgraphNode`. The following example adds both paths to a `dps::FlowgraphBuilder`.

```cpp
dps::FlowgraphBuilder theGraphB;
dps::FlowgraphNode<SplitString,MainRoute> split(theMainThread);
dps::FlowgraphNode<ToUpperCase,RoundRobinRoute> upper(processThreads);
dps::FlowgraphNode<MergeString,MainRoute> merge(theMainThread);

// Add first path through ToUpperCase
theGraphB  = split >> upper >> merge;

// Add second path bypassing ToUpperCase
theGraphB += split >>          merge;
```

The second path is added to the flow graph with the `+=` operator. The path taken is determined by compatible data object types. In this example, `CharData` will always go to `toUpperCase`, since it is incompatible with `MergeString`, which only accepts `CharData2`. On the other hand, any `CharData2` posted by `SplitString` will always go directly to `MergeString`.

### Identifying data types

When multiple paths are used in a flow graph, it is often useful to know which data types are circulating. For objects that use the `IDENTIFY` macro, the type can easily be determined by using the `is` member function. This function is a template that takes the type to check against as argument, and returns a Boolean value indicating whether the object is of the desired type. Derived types are not considered equal to their base type. The following shows an example usage case of `is`:

```cpp
CharData *cd = new CharData(), *cd2 = new CharData2();
bool a = cd->is<CharData>();     // Returns true
bool b = cd->is<CharData2>();    // Returns false
bool c = cd2->is<CharData>();    // Returns false
bool d = cd2->is<CharData2>();   // Returns true
```

### Function parallelism

The path selection mechanism described in the previous section can also be used for parallel execution of multiple functions. Figure 22 shows a simple example where `Operation1` and `Operation2` should be executed in parallel. By posting two data objects of different types, `Data1` and `Data2` from the split operation, the conditional routing will ensure both operations can be executed in parallel.
Figure 22. Function parallelism

The graph for this execution pattern is built as follows. The operations must be executed in different threads in order to ensure parallel execution. This can be achieved either by using two different thread collections (as in this example), or by ensuring that the routing functions select two different threads.

```cpp
dps::FlowgraphBuilder theGraphB;
dps::FlowgraphNode<Split,MainRoute> split(theMainThread);
dps::FlowgraphNode<Operation1,Route> op1(processThreads1);
dps::FlowgraphNode<Operation2,Route> op2(processThreads2);
dps::FlowgraphNode<Merge,MainRoute> merge(theMainThread);

// Upper path through Operation1
theGraphB = split >> op1 >> merge;

// Lower path through Operation1
theGraphB += split >> op2 >> merge;
```

Pipelining multiple operations

It is often useful to execute several operations in a pipelined parallel manner. To present this concept in our simple character processing sample, let us assume that we would like, in a second operation, to shift the character's ASCII value by a given amount, for example by one. For this purpose, we create a new operation ShiftChar, and add it to the flow graph as shown below.

```
class ShiftChar : public dps::LeafOperation<CharData2,CharData2,ProcessThread>
{
    IDENTIFY(ShiftChar);
public:
    void execute(CharData2 *in)
    {
        postDataObject(new CharData2(in->chr+1,in->pos));
    }
};
```

The ShiftChar is simply added to the flow graph between ToUpperCase and Merge. In order to ensure true pipelined parallel execution, ToUpperCase and ShiftChar obviously need to be mapped to different threads.

```cpp
dps::FlowgraphBuilder theGraphB =
dps::FlowgraphNode<SplitString,MainRoute> (theMainThread) >>
dps::FlowgraphNode<ToUpperCase,RoundRobinRoute> (processThreads) >>
dps::FlowgraphNode<ShiftChar,RoundRobinRoute> (processThreads) >>
dps::FlowgraphNode<MergeString,MainRoute> (theMainThread);
```
Do-while pipelined flow loops

In many applications, pipelining of operations is required, but the length of the pipeline cannot be determined in advance. For these situations, DPS provides the possibility to insert flow loops into the flow graph definition. A loop is defined by two parts, a condition and a target. Suppose we would like to shift the characters as long as they are smaller than 'F'. The condition is expressed as a class derived from dps::Condition that overrides the single condition member function as shown below. Again, the template parameter indicates the type of data object processed.

```cpp
class Next : public dps::Condition<CharData2>
{
    IDENTIFY(Next);
    public:
        bool condition(CharData2 *in)
        {
            return (in->chr<'F');
        }
};
```

The condition can then be inserted into the flow graph definition by using a dps::FlowgraphLoop element. The dps::FlowgraphLoop takes the condition type (Next) as template parameter, and the loop target as parameter. In this case, the loop will return to the ShiftChar node shift upon applying the condition Next. The template parameter of the condition (Next) must match the input of the preceding and following nodes (shift and merge), as well as the input of the target of the loop when the condition is satisfied (shift).

```cpp
dps::FlowgraphNode<SplitString,MainRoute> split(theMainThread);
dps::FlowgraphNode<ToUpperCase,RoundRobinRoute> upper(processThreads);
dps::FlowgraphNode<ShiftChar,RoundRobinRoute> shift(processThreads);
dps::FlowgraphNode<MergeString,MainRoute> merge(theMainThread);

// Create flow graph with a condition
dps::FlowgraphBuilder theGraphB =
    split >> upper >> shift >> dps::FlowgraphLoop<Next>(shift) >> merge;
```

A DPS loop construction does not create a cycle within the graph, but is a means of dynamically repeating at runtime certain of the graph’s operations. From the perspective of the character ‘d’, the flow graph actually looks as follows:

![Diagram](image)

Figure 24. View of the pipeline for the character ‘d’. The ToUpperCase operation is performed twice in a pipeline (no cycles in the flow graph).

Command line parameters

To facilitate testing of the application on different configurations, DPS automatically parses all command line parameters. It assumes that these parameters are in the format –key value, and stores them in an object accessible using getController()->getConfig(). The parameters can then be recovered using the getValue method. Its first parameter indicates which key must be recovered, and the second parameter indicates the default value to be used if the parameter has not been set on the command line. The type of the second parameter determines the type of the return value of the parameter.

```cpp
// Get the mapping string from the command line
const char *mapping = getController()->getConfig().getValue("proc","0 1");
if(DPS_FAILED(processThreads.addThread(mapping)))
{
    // Handle error
}
```
Full details about retrieving and querying command line parameters are available in the Reference chapter. Using the code above enables the mapping of threads to processes to be specified at launch time on the command line of the application, for example by specifying:

```
mpiexec -n 3 uppercasex -proc "0 1 2"
```

DPS provides helper objects for generating mapping strings automatically. For instance, if you use the MPI network layer, calling `dps::MPIMapper::get("2")` returns the mapping string “0 1”, while `dps::MPIMapper::get("3")` produces the mapping string “0 1 2”. If not enough MPI processes are available, process ranks wrap around, e.g. producing “0 1 0”. Other parameters may be used to produce strings that map multiple threads per process. A `PatternMapper` provides similar functionality when the TCP network layer is used.

The Reference section at the end of this manual provides full details about the use of these helper objects.

### A real application (**mandel**)

This third tutorial section presents a simple parallel application that computes images of the Mandelbrot set. It presents the concepts of complex data object serialization, flow control, and load balancing.

![Mandelbrot Set](image.png)

Figure 25. The Mandelbrot Set within the (-2-2i) - (2+2i) range. Image intensity is based on the number of iterations required before |z|>2

### Computing the Mandelbrot set

The Mandelbrot set is the set of complex numbers \( \{ c \in C \} \), where after an infinite number of applications (in the program, \( n \) applications) of the complex function \( f_c(z) = z^2+c \), the resulting absolute value \( |f_c^n(z)| \) is smaller than infinity. The Mandelbrot set is included within a region of radius 2 from the center of origin. The complex map showing the Mandelbrot set can be easily computed: we define the size of the image we would like, and then sample the area of the complex plane between (-2-2i) and (2+2i).
Application design

The mandelbrot program uses a simple split-operation-merge construct to distribute the computation of blocks of image lines across several nodes. The Split operation divides the surface for which the Mandelbrot set is to be computed into horizontal bands. The Process operation computes the iterations of the complex function, and stores the resulting bitmap in its output data object. Finally, the Merge operation collects all the resulting bitmap parts and stores the result in a bitmap file on disk.

![Flow graph for the Mandelbrot application](image)

**Figure 26.** Flow graph for the Mandelbrot application

Automatic serialization

This application introduces the automatic serialization model for data objects. Data objects using automatic serialization can contain variable sized arrays or other dynamically created structures such as matrices or image buffers. The full details are available in the Reference chapter at the end of this tutorial.

As an example, we will show the data object used to transfer parts of the Mandelbrot image, MergeInDataObject. A data object that uses automatic serialization must use several macros for declaring its data contents, as shown in the following sample:

```cpp
class MergeInDataObject : public dps::AutoSerial
{
    CLASSDEF(MergeInDataObject)
    MEMBERS
        ITEM(Int32,sizeX)
        ITEM(Int32,sizeY)
        ITEM(dps::Buffer<UInt8>,pixels)
        ITEM(Int32,firstLine)
        ITEM(Int32,lineCount)
        ITEM(Int32,taskIndex)
    CLASSEND;
};
```

The class must always start with the CLASSDEF macro, indicating the name of the type. The functionality of CLASSDEF is similar to that of IDENTIFY, therefore IDENTIFY must not be used. The list of serializable members of the class is introduced by the MEMBERS macro. Serializable members are declared with a set of ITEM macros that take the type and name of the member variable as arguments. ITEM declares public members, but specific access modifiers can be specified using PRIVATEITEM and PROTECTEDITEM (PUBLICITEM is equivalent to ITEM). The declaration of the class is completed with the CLASSEND macro.

Many types can be used in ITEM declarations. All the simple types (int, char, float, etc.) can be used, and typedefs (Int32, UInt8, Int64, Bool, etc.) are provided to specify unique type sizes for applications running on heterogeneous platforms. Standard library types (std::string, std::vector, std::set, etc.) and any other custom type that also use automatic serialization can also be used.

The MergeInDataObject for example contains a dps::Buffer for the image pixels. This is a type encapsulating a typed variable size memory block. The template parameter specifies the type of the elements within the buffer, and the block size is then given in number of elements. The block size is set in the constructor or using the resize member function. If you create a Buffer<Int32>, and call resize(4), the buffer will be large enough for 4 32-bit integers (16 bytes).

The members of the MergeInDataObject can be used as follows:

```cpp
MergeInDataObject *mit=new MergeInDataObject();

// Fill out fields of data object
mit->sizeX=IMAGE_SIZE_X;
mit->sizeY=IMAGE_SIZE_Y;
// ...
```
Flow control

When a split operation is executed, it can post a very large number of data objects. In some circumstances, this can cause major problems, for instance if the data objects are very large. Since most of the data objects posted by a split operation will be sitting in the pipeline of the following operations anyway, it is often desirable to limit the number of data objects that can be in circulation between a given split-merge pair at any time using a flow control mechanism. Implementing flow control in DPS is very simple: it is an attribute attached to a split operation node on the graph, indicating how many data objects may be in circulation at one time between a split operation and its corresponding merge operation.

For instance, in the Mandelbrot application, we might want to limit the number of data objects that can be in circulation to 3 per thread. The number of threads in the thread collection can be obtained by calling the getSize() member function of the thread collection. The flow graph builder declaration has to be changed as follows:

```cpp
dps::FlowgraphBuilder theGraphB =
dps::FlowgraphNode<Split,MainRoute>(theMainThread,0,processThreads.getSize()*3) >>
dps::FlowgraphNode<Process,RoundRobinRoute>(processThreads) >>
dps::FlowgraphNode<Merge,MainRoute>(theMainThread);
```

The additional zero before the flow control limitation in the `dps::FlowgraphNode` constructor is used for simultaneous execution. Its use will be described in the next tutorial section.

The total number of circulating data objects between a split and a merge operation does not specify how many data objects are sent to each individual thread (this is controlled by the routing functions). For example, if there are two threads in the `processThreads` collection, the maximum number of data objects circulating between the split and the merge operations is 6.

It is however possible, in situations with unbalanced tasks or processing power, for one thread to accumulate unprocessed data objects while the other threads are starved. Such cases must be handled explicitly by the routing function.

Load balancing

The Mandelbrot application provided here distributes tasks of approximately equal complexity to all computation threads. However, this is usually not the case in real applications. With a small change in the Split operation (changing a commented-out line), the Mandelbrot application presented here can be made intentionally unbalanced: the split operation produces jobs of unequal complexity. The even-numbered jobs are very small, and the odd-numbered ones are very large. Since the application distributes the jobs in a round-robin fashion to the compute nodes, a load imbalance will appear. When two processors are used, the first one will receive all the simple odd jobs, and the second one all the complex even jobs. The first processor will be sitting idle while the second one has to work through its complex jobs.

Since load-imbalance is a common problem in parallel applications, DPS provides a simple load-balancing mechanism, based on distributing new jobs to the threads which have the least number of pending tasks. The flow control system presented in the previous section allowed a limitation to be placed on the number of circulating data objects. The load balancing mechanism extends flow control by providing a routing function that will ensure that the data objects are evenly distributed on all the computation threads. In the case of the previous example, if we have a limitation to 6 data objects for 2 threads, no thread could have more than 3 pending data objects at a time.

To request load-balancing, you only need to change the routing function used in the graph declaration to `dps::LoadBalanceRoute`:

```cpp
dps::FlowgraphBuilder theGraphB =
```
Neighborhood-dependent parallel applications (*life*)

The preceding tutorials described parallel applications where the operations running in parallel in separate threads were able to process their data independently of each other. In many real applications, the processing of one part of the data is dependent on neighboring datasets, and multiple iterations are performed. Examples of such applications are finite element methods, iterative solvers, particle simulations (e.g. n-body), and cellular automata.

This tutorial presents a parallel cellular automaton. Since the same data is reused by the subsequent iterations, we use thread local variables to store groups of cells on each node. At every iteration, each cell must have the value of its neighboring cells (that may be stored on another node) before computation can occur.

The Game of Life

The game of life is a cellular automaton introduced in 1970 by the mathematician John Conway. The rules of the game are as follows:

1. The world is represented by a rectangular array of cells.
2. Every cell is either dead or alive.
3. The state of a cell in the next cycle depends on its current state as well as the states of the 8 neighboring cells according to the following rules:
   a. A living cell with 2 or 3 living neighbors remains alive, otherwise it dies.
   b. A dead cell with exactly three living neighbors becomes alive, otherwise it dies.
4. All state transitions over the entire world are synchronized.
5. The world wraps around: the column at the left side of the world is considered as the neighbor to that at the right, same for the top and bottom rows (toroidal topology).

Figure 27 shows a sample evolution sequence on a 5x5 world.

<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...O...</td>
<td>..OO.</td>
<td>..OO.</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>..OO.</td>
<td>.OOO.</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>.O..</td>
<td>.O.O.</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

Figure 27. Sample evolution sequence

Parallelization of the Game of Life

To parallelize the game of life, the world is split into horizontal tiles such that each node has a tile of equivalent size. In the processing threads, we create an array of the size of the tile plus two horizontal lines to contain copies of the bottom-most line of the tile above as well as the top-most line of the tile below.

The parallel program consists of the following stages:

1. Create the initial state of the world and distribute it to the processing threads. The initial state of the world is randomly generated.
2. Each parallel iteration is then composed of two steps which have to be executed by all tiles. First the borders are exchanged with neighboring tiles by sending the top and bottom rows to neighboring tiles, and recovering their top and bottom rows. After this border exchange, the new state of the cells in the tile is computed.
3. Once all iterations are terminated, collect all the tiles from the processing threads and store them in a file.
Figure 28. Subdivision of a 20x20 world on 4 nodes

DPS implementation of the parallel game of life

The parallel game of life uses a single flow graph, combining the three aforementioned tasks. For distributing the world across the processing threads and collecting the results after carrying out the iterations, simple split-process-merge constructs are used. The second step is more complex, since there are two split-merge constructs in sequence: the first to invoke the tile border exchange, and the second to calculate the next state of the world. The tile border exchange is implemented with another split-merge construct. The following figures show the flow graphs for the three parts of the parallel game of life: distribution of the world, evaluation of a single iteration, and retrieval of the world.

Figure 29. Flow graph segment for splitting the world across the nodes

1. SplitSendWorld
   Generate world randomly and splits it into parts sent to all processing nodes.

2. StoreWorld
   Stores received world parts as local state in thread.

3. MergeSendWorld
   Wait until all nodes have stored their part of the world.

Figure 30. Flow graph segment for collecting the world from the nodes

1. SplitReceiveWorld
   Sends requests to all nodes to read out their local world parts.

2. ReadWorld
   Sends local world part.

3. MergeReceiveWorld
   Collects all world parts into one large image.
1. **SplitToNodes**
   Split requests to start border exchange onto all processing nodes (posts empty data objects).

2. **SplitExchangeBorders**
   Split requests to transfer border parts onto the two neighbouring nodes of the current node (posts data objects containing requested border line number).

3. **CopyBorders**
   Copy requested border part into reply data object (posts a data object containing the requested border line data).

4. **MergeExchangeBorders**
   Copy border part into correct place in local storage (posts an empty data object).

5. **InnerMergeFromNodes**
   For global synchronization, wait until all nodes have finished their border exchange (posts an empty data object).

6. **InnerSplitToNodes**
   Split requests to start computation onto all processing nodes (posts empty data object).

7. **ComputeLines**
   Calculate next state of world (posts an empty data object).

8. **MergeFromNodes**
   For global synchronization, wait until all nodes have completed calculations.

---

**Figure 31. Flow graph segment for the evaluation of one iteration**

The flow graph segment that performs the iterations is placed in a DPS flow graph loop construct, allowing multiple successive iterations to be performed. The following source code shows the creation of the flow graph:

```cpp
// Split is referenced in loop, so it needs to be declared here
dps::FlowgraphNode<SplitToNodes, MainRoute> split(theMainThread);

difetimeGraphBuilder =
  // Distribute the initial world state
dps::FlowgraphNode<SplitSendWorld, MainRoute>(theMainThread) >>
  dps::FlowgraphNode<StoreWorld, WraparoundRoute>(processThreads) >>
  dps::FlowgraphNode<MergeSendWorld, MainRoute>(theMainThread) >>
  // Compute an iteration
  split >>
  dps::FlowgraphNode<SplitExchangeBorders, WraparoundRoute>(processThreads) >>
  dps::FlowgraphNode<CopyBorders, WraparoundRoute>(processThreads) >>
  dps::FlowgraphNode<MergeExchangeBorders, WraparoundRoute>(processThreads) >>
```
The flow graph is called once to perform all the tasks.

```plaintext
// Start the parallel schedule, immediately release returned data object
getController()->callSchedule(lifeGraph,new LifeDataObject())->release();
```

The routing for this application is done by using a target field in all data objects which is filled out by the functions producing the data objects. For instance, the `SplitExchangeBorders` operation sends data objects to the threads above and below the current thread. It sets the target field by using the `getThreadIndex` method of the operation to obtain the current thread index, and computes the threads above and below as follows:

```plaintext
// Request line before first line of this tile from previous tile
wt=new RequestWorldDataObject();
wt->firstLine=(getThread()->firstLine-1+worldHeight)%worldHeight;
wt->target=getThreadIndex()-1; // Previous tile
wt->lineCount=1;
postDataObject(wt);
// Request line after last line of this tile from next tile
wt=new RequestWorldDataObject();
wt->firstLine=(getThread()->firstLine+getThread()->lineCount)%worldHeight;
wt->target=getThreadIndex()+1; // Next tile
wt->lineCount=1;
postDataObject(wt);
```

The `getThreadIndex` member function of the operation returns the index of the current thread in the thread collection. Adding or subtracting one from this index respectively selects the next or previous thread in the collection (i.e. the tiles immediately below or above the local tile). The routing function finally applies the modulo operation to ensure wrap around on the top and bottom edges of the world.

```plaintext
// Wraparound routing for requests to tiles
class WraparoundRoute : public dps::Route<LifeDataObject>
{
    IDENTIFY(WraparoundRoute);
    size_t route(LifeDataObject *data)
    {
        // threadCount() is added to obtain correct result on negative inputs
        return (data->target+threadCount())%threadCount();
    }
};
```

The unfolded flow graph segment for a single iteration of the Game of Life running on three threads is represented in the figure below.
Improving overlapping of communications and computations

The computation of the central part of every band of the world can be performed without having any knowledge of the borders. This allows us to effectively remove the global barrier between the border exchange and computation of the future world state. During the border exchange, we can already compute the future state of the central bands of the world. The application flow graph would have to be changed as follows:

The implementation of this modified graph poses one additional challenge: the computation of the borders must be performed on the same thread as the computation of the center, since both computations require access to the same data. By default, DPS executes all computations assigned to the same thread sequentially. This removes any potential gains in parallelism that the optimization would provide, since the exchange of the borders and computation of the center are not executed in parallel. In order to enable true parallelism, the application can request *simultaneous execution* from DPS. This feature is enabled by passing an additional parameter to the thread collection construction, indicating how many operations may be simultaneously executing on each thread instance:

```cpp
dps::ThreadCollection<WorldThread> processThreads =
   getController() -> createThreadCollection<WorldThread>("proc", 2);
```
The above line requests two simultaneous executions, which is exactly what is required for the game of life. The operations in the flow graph can then be assigned to these simultaneous execution flows by using an additional parameter in the flow graph node construction. The first parameter of the constructor indicates the thread collection to use, and the second parameter indicates which simultaneous execution flow should be used for executing the operation.

```cpp
// Split and merge are referenced multiple times, so create them here
dps::FlowgraphNode<SplitToNodes,MainRoute> split(theMainThread);
dps::FlowgraphNode<MergeFromNodes,MainRoute> merge(theMainThread);

// Border exchange and computation on flow 0
lifeGraphBuilder +=
    split >>
    dps::FlowgraphNode<SplitExchangeBorders,WraparoundRoute>(processThreads,0) >>
    dps::FlowgraphNode<CopyBorders,WraparoundRoute>(processThreads,0) >>
    dps::FlowgraphNode<MergeExchangeBorders,WraparoundRoute>(processThreads,0) >>
    dps::FlowgraphNode<ComputeBorder,dps::NoRoute>(processThreads,0) >>
    merge;

// Center computation on flow 1
lifeGraphBuilder +=
    split >>
    dps::FlowgraphNode<ComputeCenter,WraparoundRoute>(processThreads,1) >>
    merge;
```

Only minor modifications are required on the operations in order to enable fully parallel execution. When using simultaneous execution, it is important to take into account that DPS does not perform any kind of synchronization between the multiple simultaneous execution flows. It is the programmer's responsibility to ensure coherent and thread-safe data access. The Game of Life uses two buffers for storing the world state: the first read-only buffer stores the current state, and another write-only buffer stores the future state. The two buffers are exchanged at the end of every iteration. These two buffers ensure that no race conditions can occur.
**Advanced Tutorials**

**Introduction**

The previous tutorial sections illustrated how to develop parallel applications with the DPS framework. The advanced tutorial presented in this section shows additional features of the framework, and how they can be added to the previously developed applications. The first tutorial section gives an overview of a complex DPS flow graph performing the matrix LU decomposition. The second tutorial section shows how to perform recursive parallel schedule invocations.

The remaining of the chapter describes features that are not available using the MPI networking layer. Raw TCP sockets are needed as they provide the ability to open and close connections, and start and stop processes at any time during execution (the *Reference* chapter at the end of this document describes how to launch such applications). The two subsequent tutorial sections focus on the fault-tolerance and malleability features provided by DPS. The final tutorial section focuses on parallel components and how to share flow graphs between applications.

**Complex flow graphs (lu)**

Beyond the split and merge constructs used so far, DPS provides another operation type: the *stream* operation. Stream operations can be used for improving pipelining across synchronization points. Applications often need to perform multiple computation steps, where the current computations need the results of previous computations in order to start their execution. When these computations are split up and performed in parallel, the synchronization usually is performed by ensuring that all the current computations have completed with a merge operation before splitting out the next computations.

In some cases, not all the results of the parts of the first computation are required in order to perform parts of the second computation. In these cases, it is desirable to split out these parts of the second computation before the first computation is complete in order to ensure proper pipelining of the application. The stream operation was designed to accomplish this objective, by combining the functionality of a merge operation and a split operation in a single operation.

![Figure 34. Two computations with intermediate synchronization](image)

In some cases, not all the results of the parts of the first computation are required in order to perform parts of the second computation. In these cases, it is desirable to split out these parts of the second computation before the first computation is complete in order to ensure proper pipelining of the application. The stream operation was designed to accomplish this objective, by combining the functionality of a merge operation and a split operation in a single operation.

![Figure 35. Two computations with streaming synchronization](image)

This tutorial will illustrate the use of the stream operation for computing matrix LU factorizations.

**Block-based LU factorization overview**

Block LU factorization with partial pivoting 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.

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & B
\end{bmatrix}^{r} \quad \text{where } A_{11} \text{ is a square block of size } r \times r.
\]
This matrix is decomposed as
\[
A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & X \end{bmatrix} \begin{bmatrix} U_{11} & T_{12} \\ 0 & Y \end{bmatrix}
\]

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 \( L_{11} \) is a lower triangular matrix and \( U_{11} \) is an upper triangular matrix.

Step 2. Compute \( T_{12} \) by solving the triangular system. This is the operation performed by the \text{trsm} routine in BLAS. 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 \( r \).

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

The sequence of matrix operations corresponding to these steps is shown below.

![LU Flow Graph](image)

**LU flow graph**

The matrix is distributed onto the nodes of the cluster in column blocks of size \( r \times n \). The recursion on the matrix factorization is obtained by replicating a part of the graph (in gray) once for each LU factorization level.
by using a DPS flow graph loop. The sequencing of the operations in the flow graph is accomplished by stream operations that ensure the fully pipelined execution of the application.

![Flow graph for the LU factorization. The grey part is repeated for every column of blocks in the matrix.](image)

**Figure 37.** Flow graph for the LU factorization. The grey part is repeated for every column of blocks in the matrix.

The flow graph executes the following steps:

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 notification of finished triangular system solves and stream out multiplication requests
d. parallel block-based matrix multiplication for $L_{21} T_{12}$
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.

**Recursive parallel schedules (rgraph)**

All the previous tutorials have shown applications using a single parallel schedule invocation. There are however scenarios where it is useful to use multiple parallel schedules simultaneously, for example recursive applications or hierarchical distributions. This tutorial uses a simple example of recursive invocation in order to illustrate the concepts behind multiple parallel schedule invocation.

Recursive parallel schedule invocations are ideal for some specific problem types, such as tree traversals which are commonly found in many algorithms. The split operation generates one data object for every branch on the tree leaving the current node. The leaf operation then either re-invokes the parallel schedule if the node at the end of the branch has further children, or performs the required processing. The merge operations subsequently combine the results moving back up the tree to the root node. Invoking parallel schedules is inexpensive when the same flow graph is reused. However, since all schedules will share the same thread collections, managing the routing and load balancing in recursive parallel schedules is a non-trivial task.

![Recursive invocation of a simple parallel application and b) flow graph of the resulting execution](image)

**Figure 38.** a) Recursive invocation of a simple parallel application and b) flow graph of the resulting execution

The application in this tutorial is again based on the Mandelbrot application. The general design is the same split-process-merge flow graph used previously. In this application, however, the split operation will always subdivide the received request into two subtasks containing an equal number of lines. The processing operation will examine the number of lines requested, and if it is lower than a given constant threshold, it will compute the corresponding part of the Mandelbrot set. If the number of lines is higher than the threshold, a new parallel schedule using the same flow graph will be re-invoked, and the result of this new schedule returned. The application recursively splits the tasks in two until they are smaller than a given threshold.
The leaf operation contains two parts, separated by the threshold condition on the number of lines. In order to call the recursive parallel schedule, the operation first needs to obtain a reference to the flow graph to be used. The flow graph is requested from the controller by calling its `getFlowgraph` member function with the name of the desired flow graph. The parallel schedule is then invoked by calling the `callSchedule` member function of the operation. It is important to use the `callSchedule` function of the operation rather than the `callSchedule` of the controller, since the operation's version will release the thread the operation is running on for the duration of the call. Using the controller's `callSchedule` would block the thread within the operation for the duration of the call, therefore forcing it to stay idle and potentially causing deadlocks if the called schedule needs to execute operations on that thread. If the input data object of the operation is used as input data object to the parallel schedule as well, its reference count needs to be incremented, since both the parallel schedule and the operation will release the data object. The data object returned from the parallel schedule can be used as the output data object of the operation. The process operation therefore has the following layout:

```c++
if(in->lineCount>20)
{
    // Get flow graph for recursive parallel schedule
    dps::Flowgraph fg=getController()->getFlowgraph("graph");

    // Increment reference count to input data object
    in->addRef();

    // Call parallel schedule
    ResultDataObject *mit=(ResultDataObject*)callSchedule(fg,in);

    // Post resulting data object
    postDataObject(mit);
}
else
{
    // Normal execution
    ResultDataObject *mit=new ResultDataObject();

    // ... Perform computation ...

    // Post resulting data object
    postDataObject(mit);
}
```

Fault-tolerant applications (**mandelft, lifeft**)

DPS provides fault tolerance mechanisms allowing an application to survive multiple node failures during execution. This tutorial shows how to modify the Mandelbrot and Game of Life applications to enable fault-tolerance. It only describes what needs to be modified within the application's source code and addresses some requirements imposed by the fault-tolerance mechanism. For a precise description of the inner workings of the mechanism, please read our IPDPS'05 paper, available on the `dps.epfl.ch` web site.

The fault tolerance mechanism is enabled using the `-ft` command line option. This command line option causes DPS to enable its internal fault tolerance mechanisms, in particular the duplicate data object removal. These mechanisms are not enabled by default since they induce an overhead on application performance.

When a node dies, the threads running on it are reconstructed on other nodes. This is performed through the help of backup threads. These backup threads are regular DPS threads that receive duplicates of the incoming data objects of the active threads. These duplicated data objects can subsequently be used for the reconstruction process. The location of backup threads is specified in the `addThread` function of the flow graph collection. For example, the following line adds one thread with two backup locations (on hosts `backup1` and `backup2`) to a thread collection:

```c++
threadCollection.addThread("host+backup1+backup2")
```

Specifying more than one backup thread allows more than one failure to occur, without causing more overhead: the additional backup threads are activated only when the previous backup thread in the list becomes an active thread or the previous backup thread has died.
Fault-tolerant Mandelbrot

For simple compute farm type applications like the computation of the Mandelbrot set, adding fault-tolerance with DPS requires no modification to the application code. The application has two stateless thread collections, `main` and `proc`. Fault-tolerance for the processing threads is trivial: it is sufficient to run the application with the `--ft` command line option to enable fault-tolerant behavior. When the application is running, any nodes except for the one running the `main` thread can be removed at any time without affecting the program execution.

For fault tolerance on the `main` thread, at least one backup thread needs to be added to the mapping of the thread collection `main`, as follows:

```c++
if(DPS_FAILED(mainThreads.addThread(
    getController()\rightarrow\text{getConfig().getValue("main","8000+:8001+:8002"))}))
{
    dps::DPSLog.write(0) \ll "Could not map main thread collection";
    return;
}
```

This will allow the main thread to be reconstructed on other nodes participating in the computation, ensuring successful completion if the initial main thread fails. Since the previously shown mapping provides two backups, two successive failures can be supported.

On failures, the split operation is restarted from the beginning, and all processing requests are sent again. Some of these requests will be caught by the mechanism for eliminating duplicate data objects, but since the routing function does not return constant results for a given data object when the number of total threads varies, some data objects will get routed to different nodes on re-execution, and part of the computation may be performed twice. This additional reconstruction overhead can be reduced by periodically checkpointing the main thread, i.e. by replicating its current state to the backup thread.

Checkpoining support

In order to provide support for checkpointing, the `Split` and `Merge` operations need some minor modifications, allowing them to be restarted from points other than the beginning. For the `Split` operation, the loop counter `line` needs to be made serializable within the operation, allowing the operation to be recreated from a checkpoint. This also applies to the internal output data object counter `count`. Therefore, the `Split` operation now uses the automatic serialization syntax for its data members as follows:

```c++
class Split : public dps::SplitOperation
<\text{dps::tv<SplitInDataObject>}, \text{dps::tv<SplitOutDataObject>}> 
{
    CLASSDEF(Split)
    BASECLASS(dps::OperationBase) // Ensures serialization of internal fields
    MEMBERS
        /// Current line
        ITEM(Int32,line)
        /// Current number of output data object
        ITEM(Int32,count)
    CLASSEND;
```

When the operation is initially called in normal execution, it receives an input data object as usual. However, when it is being restarted from a checkpoint after a failure, no input data object is passed as parameter (i.e. the input data object pointer is `NULL`). This particular case is used to initialize the internal variables on normal invocation:

```c++
public:
    // Split operation
    void execute(SplitInDataObject *in)
    {
        // The input DataObject is valid if the operation is not being
        // restarted, therefore we initialize our variables in that case.
        if(in)
        {
            line=0;
            count=0;
        }
```
Checkpoints may be taken when `postDataObject` is called. In order to ensure that the checkpointed state is consistent, local thread variables must be updated before the call. The updates of `line` and `count` are therefore moved.

```cpp
// Loop until all output data objects have been generated
for(; line<IMAGE_SIZE_Y;)
{
    int lc=32; // Post constant-sized chunks, 32 lines each
    if(line+lc>IMAGE_SIZE_Y)
        lc=IMAGE_SIZE_Y-line;

    SplitOutDataObject *sot=new SplitOutDataObject(line,lc,count);
    line+=lc; // Update local thread variables before postDataObject
    count++;
    postDataObject(sot);
}
```

The `Merge` operation needs similar changes in order to ensure that the current output data object state is correctly preserved when checkpointing. The output data object is stored with a `dps::SingleRef` serializable pointer.

```cpp
class Merge : public dps::MergeOperation
    <dps::tv<MergeInDataObject>, dps::tv<MergeOutDataObject> >
{
    CLASSDEF(Merge)
        BASECLASS(dps::OperationBase)
    MEMBERS
        //! The output image
        ITEM(dps::SingleRef<MergeOutDataObject>, output)
    CLASSEND;
```

Just like the `Split` operation, the merge uses the state of the input data object for initialization of the output data object. The operation ends with a call to `endSession` in the DPS controller, which causes the application to terminate. This is necessary to ensure that the parallel application terminates even when the original master node that was executing the application’s `start` method is gone. Since the application is terminated from within the merge operation, the output data object of the operation is never posted.

```cpp
void execute(MergeInDataObject *in)
{
    // If the operation is not being restarted, initialize the output image
    if(in!=NULL)
    {
        output=new MergeOutDataObject();
        output->sizeX=in->sizeX;
        output->sizeY=in->sizeY;
        output->image.resize(in->sizeX*in->sizeY);
    }
    // Wait until all image parts have been received
    do
    {
        if(in!=NULL)
            memcpy(output->image[in->firstLine*in->sizeX],in->pixels,
                   in->sizeX*in->lineCount);
    }
    while((in=(MergeInDataObject *)&waitForNextDataObject())!=NULL);

    getController()->endSession(true);
}
Finally, the application needs to call the checkpointing function for the main thread collection. Since checkpointing is a fully asynchronous process within the DPS framework, this can be done anywhere. In the present example, the Split operation is modified to request 3 checkpoints, one for every 25% of output data objects posted (assuming that flow control is used of course). We introduce an additional member variable next that indicates at which point the next checkpoint is due. This variable is checked within the main split loop, and checkpoints are requested accordingly.

```cpp
// Do some periodic checkpointing in Split
if(line>next)
{
    next+=IMAGE_SIZE_Y/4;
    // This is an asynchronous call, the checkpoint will be taken
    // shortly after.
    getController()->getThreadCollection<dps::StatelessThread>("main")
        .checkpoint();
}
```

The full source code to the Mandelbrot application with all the above-mentioned modifications is in the mandelft sample application. The code for invoking the checkpointing in the Split operation is initially commented out.

Fault-tolerant game of life

The game of life can be made fault-tolerant by following the concepts presented for the Mandelbrot previously, with one addition: since the threads are used to store a local state, this state also needs to be preserved when checkpointing. Both thread types need to be made serializable by deriving them from dps::AutoSerial and using the appropriate data declaration macros.

```cpp
class WorldThread : public dps::AutoSerial
{
    CLASSDEF(WorldThread)
    MEMBERS
        ARRAY(dps::Buffer<UInt8>,world,2)
        ITEM(Int32,firstLine)
        ITEM(Int32,lineCount)
    CLASSSEND;
};

class MainThread : public dps::AutoSerial
{
    CLASSDEF(MainThread)
    MEMBERS
        ITEM(Int32,slaveCount)
        ITEM(Int32,checkpointCounter)
        ITEM(Int64,startTime)
    CLASSSEND;
};
```

Beyond this, the other required changes are similar to those performed in the Split and Merge operations in the Mandelbrot application, i.e. adding serializable members to the operations to allow checkpointing. For very simple split operations, it may not be desirable to provide appropriate code to allow restarting from an arbitrary point. For example, the SplitExchangeBorders operation always outputs two data objects, and using a loop for this task is not elegant. Therefore, instead of using a condition and a local variable to determine which data objects need to be resent when the operation is restarted after a failure, the operation will systematically resend both, and rely on DPS's duplicate removal mechanism to remove the excess data objects. The duplicate removal mechanism relies on data object identifiers that DPS automatically assigns to posted data objects based on the order in which an operation outputs these. The current state of the data object identifier counter is preserved when the operation is restarted, causing new posted data objects to have different identifiers from the data objects posted in the previous, partial, execution. Therefore, the operation needs to explicitly specify the identifiers if it knowingly posts potentially duplicated data objects in order to ensure that the same data object always has the same identifier. The identifiers can be specified as the second argument to postDataObject. In
In the case of the `SplitExchangeBorders` operation, the first data object has a constant identifier of zero, and the second data object has a constant identifier of one. The values for the identifiers can be arbitrary, provided that a given data object always has the same identifier.

```cpp
class SplitExchangeBorders : public dps::SplitOperation
<RequestWorldDataObject, RequestWorldDataObject, WorldThread>
{
    CLASSDEF(SplitExchangeBorders)
    BASECLASS(dps::OperationBase)
    MEMBERS
        ITEM(int,active)       // Internal copies of variables taken from
        ITEM(int,iteration)    // the input data object
    CLASSEND;

    public:
    void execute(RequestWorldDataObject *in)
    {
        if(in)
        {
            // Store fields from input data object
            active=in->active;
            iteration=in->iteration;
        }

        RequestWorldDataObject *wt;
        wt=new RequestWorldDataObject();
        wt->firstLine=(getThread()->firstLine-1+worldHeight)%worldHeight;
        wt->target=getThreadIndex()-1;
        wt->lineCount=1;
        wt->active=active;
        wt->iteration=iteration;
        postDataObject(wt,0);     // Explicit constant ID = 0

        wt=new RequestWorldDataObject();
        wt->firstLine=(getThread()->firstLine+getThread()->lineCount)%worldHeight;
        wt->target=getThreadIndex()+1;
        wt->lineCount=1;
        wt->active=active;
        wt->iteration=iteration;
        postDataObject(wt,1);     // Explicit constant ID = 1
    }
};
```

The provided sample application `lifeft` implements all these changes. In order to verify that the application consistently produces correct results even in the presence of failures, `lifeft` includes a validation mechanism that verifies the results of the game of life for every operation of the parallel schedule. This type of mechanism is very useful for debugging parallel applications. In order to run the sample, the validation data must first be created by running the application once with the `-createcheck` command line parameter. After the data is created, the application can be run normally, and will use the validation data to verify its current state.

**Malleable applications (mandelm)**

DPS provides support for malleable applications by allowing thread collections to be dynamically modified at runtime. Threads can be added and removed at any time, and they can also be moved from one machine to another. Applications that do not rely on locally stored data in threads, such as the Mandelbrot application, can easily be modified to take advantage of these features. For complex applications such as the game of life, moving threads is simple, but adding and removing threads requires redistribution of the data. This tutorial section addresses the addition, removal, and movement of threads within the Mandelbrot application. Data redistribution is a complex task and is outside the scope of the present tutorial. The full source code for this tutorial can be found in the sample application `mandelm`.  

---

The case of the `SplitExchangeBorders` operation, the first data object has a constant identifier of zero, and the second data object has a constant identifier of one. The values for the identifiers can be arbitrary, provided that a given data object always has the same identifier.
Varying thread collection size

The computation thread collection used by the Mandelbrot application is composed of stateless threads, and can therefore be dynamically resized at runtime. Additional threads can be created at any time by calling the `addThread` member function of the thread collection. In order to demonstrate this functionality, we call the main parallel schedule asynchronously in the `start` function of the application, allowing for additional code to be executed simultaneously with the parallel schedule.

```cpp
// Call schedule
dps::ScheduleId sid =
    getController()->callScheduleAsync(theGraph,new SplitInDataObject);

// Perform some activity during execution of parallel schedule

// Wait for the parallel schedule to end and collect the output data object
MergeOutDataObject *ot =
    (MergeOutDataObject *)getController()->waitForSchedule(sid);
```

For the first experiment, we wait one second, and then add another thread to the processing thread collection:

```cpp
dps::SysThread::sleep(1000);        // Wait one second
processThreads.addThread(":8003");  // Add new thread
```

The new thread will immediately be used for computations, since the routing function `RoundRobinRoute` automatically distributes data objects to all available threads.

In the same way, threads can be removed at any time by calling the `removeThread` member function of the thread collection. The argument to `removeThread` is the thread identifier of the thread to be removed. Thread identifiers are constants that are used to identify the individual threads, since the index of a given thread in a collection can vary as threads are added and removed. A thread index can be converted to a thread identifier by calling the `getThreadId` member function of the thread collection. For example, we can wait one second and then remove the second thread (index of 1) from the thread collection:

```cpp
dps::SysThread::sleep(1000);        // Wait one second
// Get index of second thread and remove it
processThreads.removeThread(processThreads.getThreadId(1));
```

When a thread is removed, it will first terminate computations on all its pending data objects (i.e. the data objects that have been sent to operations on this thread).

Since modifications on thread collections only affect data objects that are posted after the modification, it is necessary to use flow control on the graph segments that run on malleable thread collections. The flow control ensures that data objects are posted continuously, rather than in a single batch at application startup. The resizing of thread collections that store local data is a much more complex process, since the data needs to be redistributed onto the new number of threads.

Moving threads

Threads can be moved at any time from one node to another by calling the `moveThread` member function of the thread collection. The `moveThread` function takes two arguments: the first is the thread identifier of the thread to move, and the second is a new mapping string for the thread. The following example will move the main thread of the Mandelbrot application from the first node (:8000) to the second (:8001):

```cpp
dps::SysThread::sleep(1000);       // Wait one second
mainThreads.moveThread(mainThreads.getThreadId(0),":8001");
```

The thread is moved at soon as the currently executing operation terminates or can be suspended. Therefore it is necessary to provide restartable split and merge operations as described in the fault tolerance tutorial section in order to ensure successful thread displacement.

Within the context of fault tolerance, `moveThread` can also be used to add backup threads to existing threads, or to change the sequence of backup thread nodes.
**Parallel components (xgraph, xgraphuser)**

DPS allows applications to share their flow graphs with other applications. The shared flow graphs typically provide services that are useful to many other applications, such as access to a parallel file system. In a parallel file system, the parallel file service can share resources, such as the disk cache, among all its clients, and potentially sequence the requests in order to optimize performance.

The DPS library provides the ability to dynamically combine flow graph parts from the various participating applications into one single flow graph. The resulting flow graph can be used to execute a single parallel schedule spanning all applications. Typical constructions combine graphs that perform specific computations to form a full application, or include a remote graph and move back and forth between the applications. This second form is typical for services, where one application makes a request to another application, and later receives the results. An example of such services is the access to a striped file spread across several nodes. Figure 39 illustrates a simple case where two applications access a striped file service. Another example is a service providing insight into the current state of an ongoing computation such as the Game of Life application presented in the previous chapter.

Using parallel components rather than libraries for services such as file I/O is useful since the parallel component can share critical resources such as an in-memory file cache amongst multiple applications or for subsequent executions of the same application.

![Diagram showing multiple applications accessing a shared parallel component](image)

**Figure 39.** Multiple applications accessing a shared parallel component

### Designing parallel component schedules

For example, the striped file service shown in Figure 39 would provide a flow graph section to applications that want to read data from a striped file. The flow graph section, shown in Figure 40, takes as input data object a request indicating which parts of the file should be read. The split operation analyzes this request and sends out individual requests to the nodes that have the data in their local storage. The data is read from the disks, and sent back in parallel to the calling application. Flow graphs that are exposed as parallel components do not need to satisfy the split-merge symmetry of complete flow graphs. The requirements need only be satisfied when the final, complete flow graph is constructed.

![Flow graph section for a striped file parallel read operation](image)

**Figure 40.** Flow graph section for a striped file parallel read operation

The flow graph section shown in Figure 40 uses two thread collections. The split operation is running on a thread collection with a single thread, with the thread local data containing sufficient information for the split.
operation to distribute the request onto the storage nodes. The read operation is running on a thread collection
with one thread per disk in the cluster. Each of these threads stores a cache of previously read data blocks in
order to accelerate future accesses. Since DPS ensures that each thread is only executing one operation at a time,
there is no problem with reentrance in the read operations even when the service is used by multiple parallel
applications simultaneously.

An application using the service can integrate this flow graph section into its own flow graph, as shown in
Figure 41. The operations belonging to the application are executed in the application's threads, whereas the
operations belonging to the parallel service are executed in the parallel service's threads. The data objects are
transparently transferred from one application to another. The use of partial graphs in parallel components is
important for optimal performance, since it allows the large data objects that result from the disk read operations
to stay within the same node for processing, as illustrated in the unfolded parallel schedule of Figure 42.

![Flow graph using a parallel service to read from a striped file](image)

---

Parallel components may also be used within the application that defines them, for example to create modular
flow graph designs.

**Working with parallel components**

Let us illustrate the implementation of parallel components by splitting up the previous Mandelbrot application
into two applications. The first application, `xgraph`, provides a computation service, where the Mandelbrot
computation is requested and performed. It contains the Split and Process operations of the Mandelbrot. The
other application, `xgraphuser`, collects the resulting Mandelbrot parts and stores the result using the Merge
operation of the Mandelbrot application. The `xgraphuser` application also creates the complete flow graph that is
subsequently used for the parallel schedule.
Figure 43. Flow graph for the xgraph and xgraphuser applications

The partial flow graphs are built in the same way complete flow graphs are created. The following code is used to create the partial flow graph in the xgraph server application:

```cpp
dps::FlowgraphBuilder theXGraphB =
  dps::FlowgraphNode<Split,dps::ZeroRoute>(mainThreads,0,10) >>
  dps::FlowgraphNode<Process,dps::RoundRobinRoute>(processThreads);

dps::Flowgraph theXGraph =
  getController()->createFlowgraph("xgraph",theXGraphB);
```

All flow graphs are automatically made available to other applications; there is no need to export them. In this example, the flow graph is available to other applications as “xgraph”. In order to use this flow graph, an application must first retrieve it, and then use it in its own flow graph building process. The flow graph is recovered using the `findFlowgraph` method of the DPS controller, which takes as arguments the name of the flow graph to be found and the network address on which to request the flow graph. The address is the one of the application instance that created the flow graph. In the present example, a simple port number is used, assuming that the server application is running on the same host as the client application. The following source code requests the parallel component flow graph, creating a local copy of the flow graph.

```cpp
dps::Flowgraph theXGraph=getController()->findFlowgraph("xgraph",":8000");
if(!theXGraph.valid())
{
  dps::DPSLog.write(0) << "Could not find remote graph \"xgraph\"\n; return;
}
```

Once the flow graph has been retrieved, it can be included within the locally constructed flow graph by using the `dps::FlowgraphSection` object. This object is similar to the `dps::FlowgraphNode` object, but takes different arguments. The two template arguments are respectively the input and output data object types for the flow graph segment used by DPS to check the validity of the final flow graph. Indeed, since the flow graph is located within another application, these types must be specified explicitly. The constructor argument for the class is the flow graph that is to be inserted. The complete flow graph of xgraphuser is therefore constructed as follows:

```cpp
dps::FlowgraphBuilder theGraphB =
  dps::FlowgraphSection<SplitInDataObject,MergeInDataObject>(theXGraph) >>
  dps::FlowgraphNode<Merge,dps::ZeroRoute>(mainThreads);

dps::Flowgraph theGraph =
  getController()->createFlowgraph("graph",theGraphB);
```

When the resulting flow graph is used to invoke a parallel schedule, DPS automatically takes care of transferring the data objects from one application to another.

An important constraint on parallel components is on the routing function for the first operation after the imported graph segment. Since the data object is routed from within the context of the application that exports the flow graph segment, it needs to be defined within that application as well as within the client application that uses the flow graph segment. In the example, inputs of the remote Split are routed using a built-in DPS routing function (`dps::ZeroRoute`), therefore available within both applications.
Simulation & Performance Prediction

Introduction

The previous sections of this manual mainly focused on the syntax used to develop parallel applications using DPS. It is however very easy to write inefficient parallel applications, either because the chosen parallelization strategy has large communication overheads, or because some of the parallelization parameters used where not optimal.

The number of parameter combinations that should be tested to reach the optimal performance of an application quickly becomes overwhelming: the granularity of the problem decomposition, the number of nodes that should be used, the mapping of threads to processing nodes, the routing functions, the use of flow control and load balancing may all play a role in reaching a good performance. To make matters worse, the optimal parameters are not only application specific, but they also depend on the speed of the network and the processing power of the cluster onto which the application will run.

This section describes how to use the benchmarking and simulation functionalities included in DPS to automatically predict the performance of DPS applications. (More information about the internal behavior of the simulator is available in our IPDPS’06 paper, available on the dps.epfl.ch website).

Simulating DPS applications

A DPS application is prepared for simulation simply by recompiling it (as well as the DPS framework) while defining the DPS_SIM symbol. Recompiled applications are then executed as usual, with the same command line parameters. The simulation is performed locally, and the predicted running time of the simulated application is outputted at the end of the execution. Note that calls to the Timer::get() method within the application return simulation times. That is, any time measurements performed within the application (e.g. to get the running time of each iteration of a computation) can be used as is.

Network and hardware parameters are specified in a configuration file (described in the following section), which is read at the beginning of the simulation. The simulator assumes that the network supports full duplex communications and that all nodes are connected to a single switch. The connector option is ignored. Application instances should be identified using the usual hostname:port syntax (except that hostnames don’t need to exist). The simulator assumes that each compute node has a single CPU. Several instances using different port number will therefore share the same processing resource.

With Visual Studio, applications can be compiled using the Sim Debug and Sim Release configurations. On UNIX systems (this includes Mac OS X and Cygwin), the recompilation is done by executing make SIM=true. On all platforms, output libraries and executables are suffixed with a ‘_s’ to clearly identify the different versions.

Cluster parameterization

The description of the network consists of a few parameters, specified in a configuration file. The default name of that file is cluster.cfg, and it can be overridden by specifying the cluster option when running the application. The required parameters are the latency of the network (in microseconds), the network bandwidth (in MB/s) and the amount of CPU usage required to send and receive data. This is provided as a list of \((n+1) \times (n+1)\) measurements, where \(n\) must be at least equal to the maximum number of simultaneous communications. The file cluster.cfg provided in the distribution contains a sample parameterization.

The DPS distribution includes the clusterParams to automatically generate such configuration files. The following options can be specified:

- map specifies the name of a file (by default nodes.map) containing a list of cluster nodes
- maxConnections specifies the maximum number of simultaneous connections (by default 1) to a same compute node. As a result, the execution will be carried on maxConnections+1 nodes, specified in the map file.
- outfile specifies the name of the file (by default cluster.cfg) to which parameters are written.
Since `clusterParams` is a regular DPS application, these options may be specified either in the configuration file or on the command line. All deployment options (e.g. the node names, port numbers, connector choice) are identical to the ones of any DPS application.

**Direct execution simulation**

By default, the simulator executes the actual application code and measures the running time of each operation. The advantage of this method is that applications can be simulated without modifying their source code. It has the following disadvantages however:

- The simulation must run on the same hardware than the one used to run the real parallel application (but only a single node is needed).
- The memory consumed by the simulation is the sum of the memory consumed by the application on every processing node during a real execution.
- The running time of the simulation can be as large as the parallel running time multiplied by the number of processing nodes used by the application.

As a result, the size of the applications that can be simulated through direct execution is limited. The following sections describe how to overcome these limitations.

**Partial direct execution simulation**

For most applications, the parallelization pattern (e.g. the number and size of transferred data objects) is independent of the actual computation results. For instance, the number of block multiplications performed in a matrix multiplication application only depends on the number of blocks, and not on their content. When that is the case, computations do not need to be performed, and can be replaced by a notification of their running time to the simulator. Doing so not only reduces the time and memory required by the simulation, but it can also make the simulation portable. Indeed, running times may now be specified independently of the hardware on which the simulation is running.

Let us take an operation that multiplies a matrix stored in its thread with the one received in a data object:

```c
void execute(MatrixDataObject *in)
{
    MatrixDataObject *out;
    // Output matrix is allocated by multiplication
    out->matrix=matmul(getThread()->matrix, in->matrix);
    postDataObject(out);
}
```

Now let us assume that we have found a model that predicts the running times of the `matmul()` function on this particular hardware for matrices of size $n \times n$: $t(n) = 2.13n^3 + 3n^2 + 1.24n$

Assuming that all matrices are square, we can use this model to inform the simulator of the running time of the skipped computation using the `addComputationTime` method, which takes a number of microseconds as sole parameter.

```c
void execute(MatrixDataObject *in)
{
    MatrixDataObject *out;
    #ifdef DPS_SIM
    out->matrix=matmul(getThread()->matrix, in->matrix);
    postDataObject(out);
    #endif
    // Skip computation
    int n=getThread()->matrix->size;
    addComputationTime(2.13*n*n*n + 3*n*n + 1.24*n);
    // We must now allocate a matrix, as matmul would have done
    out=new MatrixDataObject(n);
    #endif
    postDataObject(out);
}
```

It is quite rare that we are able to approximate a computation using a simple polynomial. When more complicated models are used, or when the running time is given by an external source (e.g. a file or a hardware simulator), the running time prediction may be distorted. Indeed, calling `addComputationTime` does not stop the
simulator from measuring the operation running times. We may therefore need to correct the prediction as follows:

```c
void execute(MatrixDataObject *in)
{
    MatrixDataObject *out;
    #ifndef DPS_SIM
        out->matrix=matmul(getThread()->matrix, in->matrix);
    #else
        Int64 start=dps::Timer::get();
        Int64 computationTime=getPredictionFromModel(in->matrix);
        Int64 overhead=dps::Timer::get()-start;
        // Subtract time required to obtain prediction
        addComputationTime(computationTime-overhead);
        // We must now allocate a matrix, as matmul would have done
        out=new MatrixDataObject(in->matrix->cols);
    #endif
    postDataObject(out);
}
```

Note that it is perfectly possible to select which operations should be performed (to specify the destination thread of a data object for instance), and which may be skipped.

Direct execution and partial direct execution can coexist within the same application, allowing the comparison of the prediction results of each method. An example is given in a following section that illustrates how to choose the simulation method with a command line option.

### Automated benchmarking

If no model is available, we can simply benchmark the various operations on the real hardware, and reuse the measured times when simulating the application. DPS provides a few helper classes to automate that process.

### Collecting measurements

Measurements are collected using the `BenchWriter` class. The file to which measurements are appended is specified as a parameter to the `BenchWriter::init` method. Cleanup is performed by calling `BenchWriter::finalize`.

```c
// Before the first measurement is taken
dps::sim::BenchWriter::get()->init("measures.csv");
// Before the application terminates
   dps::sim::BenchWriter::get()->finalize();
```

The `BenchWriter` class acts as a stop watch, where the timer is started by calling `start`, and calling `write` causes the time elapsed since the last call to `start` to be written to the file specified when `init` was called. The `write` method may take between one and four `double` parameters in addition to the name that identifies the measured function.

```c
dps::sim::BenchWriter::get()->start();
// Do things during 123000 microseconds
   dps::sim::BenchWriter::get()->write("funcName",12,3.45);
```

The following line is then added at the end of the output file:

```
funcName,12.000,3.450,123000
```

The function name and the parameters form a single key that identifies the measurement. As a result, the parameters of `write` should be the parameters that influence the duration of the computation enclosed within the calls to `start` and `write`.

There is obviously no strict rule as to which part of an operation should be measured. In some cases, one might prefer to measure the running time of several functions separately, while in other cases it is sufficient to measure the running time of the whole operation at once. However, operations are suspended during calls to
**postDataObject** and to **waitForNextDataObject**. Measurements should therefore not include any of those two functions.

A simple way of collecting measurements is to instrument the application code. That is, we run the application as usual, and measurements for the corresponding parameters are taken. This ensures that all relevant measurements are taken, and automatically causes more measurements to be taken for frequently occurring computations.

The instrumented application should be running within the simulator, which guarantees that no two operations ever run simultaneously, and that writes to the file never overlap. If you really need to measure running times using the real parallel application (to speed up the process for instance), make sure of the following:

- No two operations may run simultaneously on the same compute nod. A simple solution is to map each thread on a different node. This ensures that the processing power is never shared between operations during the measurements, and that calls to write match the corresponding call to start. There is no support for simultaneous execution.
- No two operations that may run simultaneously on different machines write their data to the same file. Here again, one solution is to open one file per thread, e.g. by calling **BenchWriter::init** in the thread constructor and **BenchWriter::finalize** in the destructor.

### Reusing measurements

The process of retrieving the measurements from a data file is quite similar, and is handled through the **Bench** class. The call to **init** aggregates the data read in the specified file by averaging all the measurements for every key, i.e. for every different value of “function name + parameters”. The timings may then be retrieved using the **Bench::getTiming** method, which takes the same parameters as **BenchWriter::write**.

```cpp
dps::sim::Bench::get()->init("measures.csv");
double t=dps::sim::Bench::get()->getTiming("funcName",12,3.45); // t equals 123000
```

Since the averaging process may take some time when the file contains lots of measurements, aggregated results are stored in a new file with a .agg extension which is reused by future simulations. (When more measurements are collected the file containing the aggregated results is emptied to ensure that aggregate values are recomputed when the next simulation is run.)

### An example

The DPS distribution contains two example applications that integrate measurement and simulation capabilities: **matmul-sim** (parallel matrix multiplication) and **lu-sim**. The latter is identical to the LU factorization application **lu** described in the Advanced Tutorial section.

When compiled without defining DPS_SIM, both applications execute normally. When DPS_SIM is defined, one of the following options may be specified on the command line or in the configuration file: **directExec** to run a direct execution simulation, or **simBench** to collect running times. If no option is specified, the simulation uses running times retrieved using the **Bench** class.

The following code from the InitialSplit operation of **lu-sim** integrates all the notions described so far in this section.

```cpp
// Original code for InitialSplit split operation
void execute(RoutedDataObject *in)
{
    LUApp *app=(LUApp*)getApplication();
    if(in)
    {
        Matrix sm;
        sm.set(getThread()->local,0,0,app->b,getThread()->local.rows);
        float d;
        if(sm.cols%SBSIZE!=0)
            ludcmp(sm,&d,getThread()->vv);
        else
            blockludcmp(sm,&d,getThread()->vv,SBSIZE);
    }
}
```
This operation spends most of its time performing the LU decomposition of a block stored in the local state (\textit{ludcmp} or \textit{blockludcmp}). We therefore remove this operation, while the loop that posts the output data objects is left as is.

```cpp
for(int i=1;i<getThread()->blocksX;i++)
{
    MatrixDataObject *lur=new MatrixDataObject();
    lur->matrix.set(getThread()->local,0,0,app->b,app->b);
    lur->route=i;
    lur->level=0;
    lur->blocksX=getThread()->blocksX;
    postDataObject(lur,i);
}
```
Avoiding memory allocations

When the simulation runs without performing actual computations, it may become possible to avoid the allocation of certain parts of thread states and of data objects. Going back to our block matrix multiplication example, it is clear that if no computation is performed, the actual content of the matrix does not matter to the evolution of the computation, and that there is therefore no need to allocate the matrix.

The simulator still needs to compute the size of data objects to determine their network transfer time. It does so without accessing the actual memory content however, so the size computation remains correct as long as the description of the data object is correct. The following code snippet from the parallel Mandelbrot computation (see the “A real application (mandel)” section) illustrates this principle using the `Buffer::simResize` function:

```c++
#ifndef DPS_SIM
   // Allocate enough space for pixels in output data object
   mit->pixels.resize(in->lineCount*IMAGE_SIZE_X);

   // Fill out pixels in output data object
   for(Int32 j=0; j<in->lineCount; j++)
   {
      for(Int32 i=0; i<IMAGE_SIZE_X; i++)
      {
         mit->pixels[i+j*IMAGE_SIZE_X]=mandelbrotFunction(i, j+in->firstLine);
      }
   }
#else
   // Pretend to resize the buffer
   mit->pixels.simResize(in->lineCount*IMAGE_SIZE_X);
   // Retrieve computation time
   addComputationTime((Int64)dps::sim::Bench::get()->getTiming("mandel",
                    in->lineCount,IMAGE_SIZE_X);
#endif
```

The `Buffer::simResize` function updates the variable that stores the buffer size without performing any allocation. The computation of the data object size therefore remains correct. (Copy constructors and operator overloads of the `Buffer` class assume that memory is properly allocated. They should therefore all be replaced by explicit calls to `simResize` in order to prevent runtime crashes.)

Running `lu-sim` with the `noAlloc` option prevents it from allocating matrices, which greatly reduces the memory consumption of the simulation and reduces its running time, while having little effect on the runtime prediction. Regarding the implementation, we simply needed to adapt the Matrix class to allow unallocated matrices, and remove/replace calls to `Matrix::resize` and to `memcpy`. Detailed prediction results and performance measurements are available on the DPS website.
Message Race Detection

Introduction
Writing an efficient parallel application often requires minimizing the amount of synchronization enforced between the different compute nodes. Within DPS applications, this translates into avoiding global merge-split pairs of operations, or in their replacement by stream operations. Removing synchronizations is equivalent to removing relative ordering constraints between operations. If too much synchronization is removed and the execution ordering of non-commutative operations is no longer imposed by the flow graph, the computation may produce incorrect results. Since DPS operations are triggered by the delivery of a data object (or message), we refer to such occurrences as message races, following standard naming conventions.

Such synchronization errors often appear only in specific cases, when for instance subtle timing differences delay the delivery of one data object a bit more than usual, causing another operation to start in the mean time. The first difficulty therefore lies in being able to reproduce the errors, such that they can be identified and corrected.

The two figures below illustrate such a situation. Figure 44 shows the flow graph of a Game of Life application, where the central synchronization has been removed (the original flow graph is displayed in Figure 32). The consequences of this apparently minor modification are hard to detect at first sight, but this change is sufficient to allow the situation illustrated in Figure 45. If for some reason the data object sent from the thread Main to the thread Proc 1 is delayed, the exchange border requests are sent after the computation of the new thread state on Proc 0 and Proc 2. Proc 1 therefore receives updated borders from its neighbors, which means that its computations will produce incorrect results. Since multiple iterations are performed, the error will spread to the whole world, potentially leading to a completely different outcome.

![Figure 44. Flow graph for the Game of Life without intermediate synchronization.](image1)

![Figure 45. If the execution of the split operation on Proc 1 is delayed, Proc 0 and Proc 2 may send their borders to Proc 1 after they have been updated, thereby producing incorrect results.](image2)
The DPS framework includes a testing tool that automatically executes all possible orderings of the messages sent by a parallel application, and detects differences in computation results. Since the number of orderings that must be executed quickly explodes, a novel decomposition technique was developed to drastically reduce the time required to test all orderings. This section explains how this tool can be used, and which applications can take advantage of it. If you want to know more about its internal behavior or see results about its efficiency, read our IPDPS’07 paper, available on the dps.epfl.ch website.

What does the tool do?
The testing tool first executes the parallel application once and captures all the operations and the data objects of the computation. It then builds a dependency graph that describes the partial orderings between all the operations performed and the data objects sent. The analysis of the graph then reveals sets of operations that do not interact with each other.

Within each set, we then reexecute all the operations in different orders, and for each one of them, the tool checks that:

1. The data objects generated are identical to the ones generated during the reference execution
2. The final state of each thread is the same after the execution of every ordering of operations.

If an ordering causes one of these tests to fail, a binary file containing the partial application trace is produced. The initial ordering and the one that produced a different outcome are also stored, so that they can then be replayed and compared in order to determine the origin of the error.

It is important to note that the results of a test only apply to the particular input that was tested. For many applications, testing one or a few sets of inputs is sufficient to ensure that there are indeed no possible message races. In the general case however, an application may use different execution paths depending on the data being processed. All such cases must be therefore tested separately.

Application requirements
As one can guess from the previous section, the tested application must produce a fixed set of data objects. That is, the number of data objects sent during the execution of the application as well as the content of these data objects must be independent of the ordering of their delivery (it may obviously depend on the input data of the application). Other applications may still be tested, but results will in most cases be meaningless.

One family of problems that do not produce a fixed set of data objects consists of divide-and-conquer optimization problems, where at any time the current best solution found is used to speed up the remainder of the computation. In such cases, updates to the best solution found so far are often broadcasted to all the compute nodes in order to reduce processing times. Changing the processing order may therefore cause a good solution to be detected more or less early in the computation, thereby resulting in different update data objects to be sent. Note that non-deterministic applications are sometimes made deterministic during development and debugging phases in order to increase the reproducibility of the computations. Such applications may therefore also be tested.

Thread states must be serializable
In order to compare thread states and data objects, we must be able to checkpoint thread states, and all serializable objects must derive from dps::AutoSerial. See the Reference chapter and the Fault-tolerant game of life tutorial for more details. (Operations are never checkpointed or restarted, and therefore do not need to be serializable.)

Serializable object members must always be initialized
Ensure that all members of all serializable objects are initialized upon construction, or before being posted. Failing to do so causes uninitialized (and most likely different) values to be compared, causing the testing tool to report spurious errors.

Within operations, use GETTHREAD(member) instead of getThread()->member
The GETTHREAD macro tells the testing tool which members are read by each operation, thereby allowing further optimizations to be performed.

The testing tool is built on top of the simulator. The application must therefore also be compiled with the DPS_SIM symbol, using for instance the “Sim Debug” and “Sim Release” configurations in Visual Studio or
Controlling the testing procedure

Recompiled applications are executed as usual, with the same command line parameters. The testing tool is simply activated by adding a -sim.verify flag on the command line. During execution the information about which thread, and which subgroup of operations on this thread is being tested is displayed. The decomposition into operation subgroups greatly reduces the number of operations in each subgroup, and therefore the number of orderings that must be tested for every subgroup. For complex applications however, some sets of operations may still allow a prohibitive number of orderings. This can also occur when an application runs on many machines, as for instance all the permutations of inputs of merge operations must be tested. The testing tool therefore provides several ways to execute only a subset of all possible orderings.

-sim.verify static: this is the default mode. The tester executes all orderings if the number of operations in the subgroup is smaller than a predefined threshold and generates a set of orderings that have a high probability of revealing existing messages races otherwise. The ordering generation algorithm is described in our IPDPS’07 paper. The threshold is a constant specified in the file validationutils.cpp.

-sim.verify staticFull: forces the execution of all possible orderings.

-sim.verify staticRandom: behaves like the default mode, except that orderings are generated randomly when the threshold is exceeded. The number of random orderings to be generated is specified next to the threshold in the file validationutils.cpp.

Replaying an erroneous execution

Once a different message or thread state is found, the testing tool dumps its state into .subgroup file and stops its execution. The execution can simply be replayed by executing the application with the flag -sim.replay filename.subgroup. Since a single application instance is used, the replayed execution may easily be run within a conventional debugger.

Run the demo

The lifev directory included in the DPS distribution includes two versions of the parallel Game of Life that have been updated to support the features described in this section. The lifev-singlebuf.cpp file does not double-buffer the world part on each thread, and is therefore subject to the message race illustrated in Figure 45 if the incorrect flow graph is used, i.e. when the -wrong flag is specified on the command line. The following commands test the erroneous case:

```
make SIM=true
cd bin
./lifev-singlebuf_s -wrong -sim.verify
```

The execution stops when the race is detected on thread proc[0], i.e. when a CopyBorders operation sends a different data object. The tool displays the difference between the expected and the sent data objects, and dumps all necessary information into the replay file LifeApp_WorldThread[0].subgroup. The replay may then be performed using:

```
./lifev-singlebuf_s -wrong -sim.verify \
-sim.replay LifeApp_WorldThread[0].subgroup
```
Reference

Introduction
This section presents in detail how to use the DPS classes and configure the DPS runtime system. It addresses various parts of DPS in depth, such as the configuration mechanisms, the creation of thread mappings, and data serialization.

Command line parameters and configuration file
When the number of parameters grows, it becomes more convenient to store them into configuration files than to always specify all parameters on the command line. The DPS library provides a simple interface for handling both methods of input. The command line options and the configuration file are combined into a single configuration set. For example, the following configuration file and command line are equivalent:

Command line:

```
./MyApp -strparam "Some text" -intparam 2 -realparam 3.0 -flag
```

Configuration file:

```
# Configuration file, # is for comments
strparam=Some text
intparam=2
realparam=3.0
flag=    # No value
```

When two options with the same name appear on the command line and in the configuration file, the command line takes precedence. The name of the configuration file to be loaded can be specified on the command line with the -cfg option. The default configuration file is `dps.cfg`.

Using the configuration parameters
The configuration options can be retrieved by calling the `getConfig` method of the DPS controller. `getConfig` return an instance of `dps::StringMap`, which is a collection of string pairs: the options and their values. There are three basic methods for getting option values from the string map, depending on their type:

```
dps::StringMap& config = getController()->getConfig();
// String values
const char *strValue = config.getValue ("strparam","default");
// Integer values
int intValue = config.getValue("intparam",1);
// Floating point values
double realValue = config.getValue("realparam",1.0);
```

For all of these calls, the first parameter is the name of the option, and the second parameter is the default value that is returned when the option is not specified. To determine whether an option is set, the `isSet` method can be used.

```
bool set = config().isSet("flag");
if (!config.isSet("intparam"))
    printf("Please specify a value for intparam\n");
```

This call returns true if the option is set on the command line or in the configuration file. In some cases, it is useful to group the options for a particular part of the application, e.g. to avoid collisions within the option names. DPS directly supports this by providing support for hierarchical option names. For example, all the options for DPS networking start with `net.: net.port, net.address, net.nagle`. The configuration
set can then be queried for a subset consisting of only those options starting with a specific string, as shown in the following example.

```cpp
dps::StringMap subConfig;
getController() ->getConfig().getSubset("net", &subConfig);
int port = subConfig.getValue("port", "5005"); // Queries 'net.port'
```

### The DPS Controller

DPS uses a `Controller` object to manage all the resources used by a parallel schedule: the thread collections and the threads they contain, the flow graphs, and the state of the currently executing parallel schedules. The Controller also owns some helper objects, such as the network layer used for communicating with other nodes, and the connector used to execute remote instances of the parallel application.

The controller object also exposes the interfaces to the DPS library for tasks such as creating thread collections, creating flow graphs, or executing parallel schedules. Most DPS objects provide a method to obtain a pointer to the application's controller:

```cpp
dps::Controller *controller = getController();
```

### The `dps::Application` class

DPS applications derive from the `Application` base class, which contains the startup functions of the application. DPS provides two startup functions, `init` and `start`. The `init` function is called on all application instances, whereas `start` is only called in the first instance of the application. Both functions are called after the initialization of the DPS controller, and can therefore access all the features of the DPS library. Like all classes used within the DPS framework, the `IDENTIFY` macro is used to provide basic type reflection features.

```cpp
// Simple application class sample
class SimpleParallelApp : public dps::Application
{
    IDENTIFY(SimpleParallelApp);
public:
    // Application startup function
    virtual void start();
    // Application initialization function
    /* Returns true if successful, false if initialization has
    failed. DPS will quit on failure.
    */
    virtual bool init();
    // Application help string
    virtual std::string help();
};
```

The startup function `start` typically contains the code required to create the thread collections and the flow graph, and is also responsible for running the parallel schedule. The `init` function is convenient for initializing global variables or do some parameter checking before any operation starts running. It returns `true` if the initialization is correct and `false` otherwise. In the latter case, the application instance is shut down. Here is an excerpt from the `life` program:

```cpp
virtual bool init()
{
    worldWidth  = getController() ->getConfig().getValue("w", 400);
    worldHeight = getController() ->getConfig().getValue("h", worldWidth);
    return ((worldWidth > 0) && (worldHeight > 0))
}
```

The help function is called when the `-help` option is specified on the command line. From the `life` program:

```cpp
virtual std::string help()
{`
The application class is also an ideal place to store globally accessible variables, since a pointer to the application object can easily be obtained from within the context of most other DPS objects through a `getApplication` member function.

## Threads & thread collections

Threads are user-defined C++ classes representing the thread local storage. A thread collection is a variable-sized distributed container of threads. If the operations running within a thread collection do not require access to any locally stored data, DPS provides a specific thread class `StatelessThread` that can be used to indicate that the threads do not store any local state.

Creating a simple stateless thread collection requires only a single parameter: the name of the thread collection.

```c++
dps::StatelessThreadCollection processThreads =
  getController()->createStatelessThreadCollection("process");
```

In order to create a more complex thread collection controlling the aggregation, the operating system thread mapping and the thread type, several additional arguments can be specified. The thread type is provided as a template argument in order to allow validation of the type at compilation time.

```c++
dps::ThreadCollection<StorageThreadType> processThreads =
  getController()->createThreadCollection<StorageThreadType>(
    "process", // name
    2,         // Number of OS threads per DPS thread (default 1)
    false     // Enable aggregation (default true)
  );
```

The `ThreadCollection` object provides methods for managing the threads it contains. Threads can be added, removed and moved from one node to another. Before the thread collection can be used for executing parallel schedules, it needs to contain at least one thread. The following methods are used for updating the thread collection:

```c++
// Add a new thread
processThreads.addThread("hostname");

// Remove a thread
processThreads.removeThread(threadId);

// Move a thread
processThreads.moveThread(threadId,"newhostname");
```

Since threads may be added and removed, their index within the thread collection may change over time. The `removeThread` and `moveThread` methods therefore use the threads’ unique identifiers. The thread collection provides two functions to convert from identifiers to indices and vice versa.

```c++
// Convert from identifier to index
size_t threadIndex = processThreads.getThreadIndex(threadId);   

// Convert from index to identifier
ThreadId threadId = processThreads.getThreadId(threadIndex);
```

## Generating thread mapping strings

The thread mappings specified as input parameters to the `addThread` and `moveThread` methods are strings containing a list of host names. Rather than specifying the mapping within the application source code, they may be read from the configuration file or from the command line using the `getConfig` member of the DPS controller:
processThreads.addThread(getController()->getConfig().getValue(  "proc",  "host1 host2 host3"));

The above line will use the value of the command line parameter proc as a mapping string. If the parameter is not specified on the command line, it will attempt to find the value in the DPS configuration file. As a final fallback, the set of hosts "host1 host2 host3" will be used.

TCP network layer

DPS also provides a mechanism to automatically generate the thread mapping strings based on a list of machines, the pattern mapper. The pattern mapper reads a list of nodes from a file, and assembles these nodes into a mapping string according to a specific pattern. The mapping pattern has the following format:

<nodes>[x<multiple>][+<offset>][b<backups>]

The initial element indicates the number of nodes to use. The multiple indicates how many times each node should be used. Therefore, the resulting size of the thread collection will be <nodes> multiplied by <multiple>. The offset indicates how many hosts should be skipped in the host file before starting the mapping. The offset can be used for example to create disjoint thread collections. The backup threads are used by the fault tolerance mechanism. The following source code shows how the pattern mapper is typically used, with both the name of the host file and the pattern being specified on the command line:

```cpp
// Create a PatternMapper object
dps::PatternMapper pm
  (getController()->getConfig().getValue("map", "nodes.map"));
// Add threads to thread collections using mappings generated by the
// PatternMapper
theMainThread.addThread
  (pm.get(getController()->getConfig().getValue("mpat","1")).c_str());
processThreads.addThread
  (pm.get(getController()->getConfig().getValue("pat","3")).c_str());
```

The PatternMapper object is constructed with a file name as argument. This file is a simple list of host names to use, with one name per line. In this example, the name of the host file is specified in the command line parameter map. These hosts are subsequently combined into a mapping string by calling the get method of the PatternMapper. The get method takes a single argument, which is a mapping pattern, and generates an output string as a std::string object. Here are a sample nodes.map file and a few mapping strings returned by the pattern mapper:

```
host1
host2
host3

// Three threads
pm.get("3") returns: "host1 host2 host3"

// Four threads, two on each host
pm.get("2x2"): host1 host1 host2 host2

// Four threads, two on each host, offset by one
pm.get("2x2+1") returns: "host2 host2 host3 host3"

// Three threads with two backups
pm.get("3b2"): host1+host2+host3 host2+host3+host1 host3+host1+host2

// Two threads on one node with one backup, offset by two
// (backup threads wrap to first node)
pm.get("1x2+2b1"): host2+host3 host2+host3
```
**MPI network layer**

DPS provides a similar mechanism for generating mapping strings with MPI process ranks. Since these are identified using simple integers, no external list of machines needs to be loaded. The `MPIMapper` uses patterns have the same form as the ones presented above for the `PatternMapper`, with the exception that it does not support backup threads.

```cpp
// Three threads
dps::MPIMapper::get("3") returns: "0 1 2"

// Four threads, two on each host
dps::MPIMapper::get("2x2") returns: "0 1 0 1"

// Four threads, two on each host, offset by one
dps::MPIMapper::get("2x2+1") returns: "1 2 1 2"
```

**Flow graphs**

Flow graphs are built by combining the operations, routing functions, conditions, and thread collections using a flow graph builder. The C++ classes used for the operations, the loop conditions and the routing functions are inherently static, since they are compiled by the C++ compiler. On the other hand, the flow graph creation process is handled at runtime, and therefore provides great flexibility in the assembly of the various elements. The flow graph builder is implemented in the `FlowgraphBuilder` object, which is used to combine sequences of operations into a single graph. The operations within the flow graph are described by a `FlowgraphNode` object. The `FlowgraphNode` object encapsulates the operation class, the thread collection within which the operation is to execute, and the routing function for selecting the target thread within the thread collection. A sequence of operations is created by combining several `FlowgraphNodes` with the >> (right shift) operator, and added to the `FlowgraphBuilder` with the += operator. These overloaded operators provide a natural syntax for describing sequences of operations.

![Flow graph diagram](image)

**Figure 46.** Simple flow graph for a compute farm, showing operations, thread collections, and routing functions.

For example, to create the flow graph illustrated in Figure 46, the following source code would be used:

```cpp
// Declare the flow graph nodes (operations)
dps::FlowgraphNode<Split, dps::ZeroRoute> split(masterThreads);
dps::FlowgraphNode<ProcessData, dps::RoundRobinRoute> process(processingThreads);
dps::FlowgraphNode<Merge, dps::ZeroRoute> merge(masterThreads);

// Create a flow graph builder object
dps::FlowgraphBuilder graphBuilder;

// Add a chain with the three operations
graphBuilder += split >> process >> merge;
```

Template parameters to the flow graph nodes specify the elements that are statically known at compilation time, such as the operation type and routing function type. These elements are used by DPS to perform some validation of the flow graph at compilation time to ensure that the flow graph is valid. DPS verifies the following points:
• The type of the threads in the thread collection specified for a FlowgraphNode is the type specified for the operation's execution thread. For operations that do not specify a specific thread type, any thread type is acceptable.

• The operation and the routing function specified for a FlowgraphNode contain the IDENTIFY macro, allowing them to be created by the DPS library.

• The operation before the >> operator has at least one output data object type in common with input data object types of the following operation. This check ensures that the two operations can effectively communicate with one another.

• When a loop construct is inserted, the input data object of the first operation enclosed within the loop is the same as the output data object of the last operation, in order to ensure that the output data object can be looped back as input. Another constraint that must be satisfied is that the flow graph keeps the symmetry between split and merge operations. The latter can only be checked at runtime however, when the condition that determines the number of repetitions is evaluated.

In order to create complex flow graphs containing multiple branches, multiple sequences of operations can be added to the flow graph builder.

![Flow graph with multiple branches](image)

Figure 47. Flow graph with multiple branches

For example, to create the flow graph illustrated in Figure 47, the following source code would be used.

```cpp
// Declare the flow graph nodes (operations)
dps::FlowgraphNode<Split, dps::ZeroRoute> split(masterThreads);
dps::FlowgraphNode<ProcessData1, dps::RoundRobinRoute> process1(processingThreads);
dps::FlowgraphNode<ProcessData2, dps::RoundRobinRoute> process2(processingThreads);
dps::FlowgraphNode<Merge, dps::ZeroRoute> merge(masterThreads);

// Create a flow graph builder object
dps::FlowgraphBuilder graphBuilder;

graphBuilder += split >> process1 >> merge; // Sequence for upper path
graphBuilder += split >> process2 >> merge; // Sequence for lower path
```

Since the flow graph nodes split and merge are used twice, the flow graph builder knows that the operations appearing between them are on different paths between these two flow graph nodes. This process can be extended to an arbitrary number of branches within the flow graph.

![Complex flow graph with multiple branches](image)

Figure 48. Complex flow graph with multiple branches

The flow graph shown in Figure 48 can be created as follows (the declaration of the flow graph nodes has been omitted):

```cpp
graphBuilder += s1 >> s2 >> o1 >> m2 >> m1; // First branch
graphBuilder += s2 >> o2 >> m2; // Branch with o2
graphBuilder += s1 >> o3 >> m1; // Branch with o3
```
Only the parts of the flow graph that have not yet been specified need to be added to the flow graph builder. In the above example, the path from \( s1 \) to \( s2 \) is only added once. Flow graphs of arbitrary complexity may be constructed by using this mechanism. Subsequent flow graph nodes may for instance be added using a for loop, allowing the flow graph to be constructed at runtime to meet the application's requirements. Once the flow graph has been completely described within the flow graph builder, the final flow graph object is created by the controller.

```cpp
dps::Flowgraph graph =
  getController() -> createFlowgraph("graph", graphBuilder);
```

Once the controller owns the flow graph, it can no longer be modified. However, the same operations, routing functions and thread collections can be reused in any number of distinct flow graphs.

**Operations**

The nodes of a flow graph are its operations. Within DPS, all operations share a common syntax and must derive from one of the following base classes: LeafOperation, SplitOperation, MergeOperation or StreamOperation. The base class is a template, taking as arguments several of the operation's attributes: the input data object type, the output data object type, and the thread type on which the operation executes. The template arguments are used for validating the sequence of operations within a flow graph at compile time. Like all classes used within the DPS framework, the IDENTIFY macro is required.

The main body of the operation is located in the `execute` member function. This function is called by the DPS library when a data object needs to be processed by the operation. An operation can contain any amount of C++ code. When an operation wishes to post a new data object, it can call the `postDataObject` function provided by the base class.

A leaf operation is described as follows, and must post exactly one output data object:

```cpp
class UserLeafOperation : public dps::LeafOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  IDENTIFY(UserLeafOperation);
  void execute(InputDataType *in)
  {
    /* Perform some processing */
    postDataObject(new OutputDataType());
  }
};
```

Split operations may generate any number of data objects, typically using for or while loops.

```cpp
class UserSplitOperation : public dps::SplitOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  IDENTIFY(UserSplitOperation);
  void execute(InputDataType *in)
  {
    // Post 10 output data objects
    for(int i=0; i<10; ++i)
    {
      postDataObject(new OutputDataType());
    }
  }
};
```

Merge operations are implemented in a similar fashion, with the main difference that they need to receive multiple input data objects. Subsequent data objects are retrieved using the `waitForNextDataObject` function. This function returns NULL if all the data objects have already been processed. Like leaf operations, the merge operation must post exactly one data object. It should also loop on `waitForNextDataObject` until NULL is returned in order to ensure that no orphan data objects are left within the parallel schedule.

```cpp
class UserMergeOperation : public dps::MergeOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  // Merge operations
  IDENTIFY(UserMergeOperation);
  void execute(InputDataType *in)
  {
    // Merge data objects
    while (true)
    {
      // Post data objects
      postDataObject(new OutputDataType());
    }
  }
};
```
A stream operation uses the same general structure as a merge operation, but it may contain multiple calls to `postDataObject` like split operations. The following source code segment illustrates a typical stream operation:

```cpp
class UserStreamOperation : public dps::StreamOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  IDENTIFY(UserStreamOperation);
  void execute(InputDataType *in)
  {
    // do-while loop until waitForNextDataObject returns NULL
    do
    {
      /* Do something with data object in */
      } while((in = waitForNextDataObject())!=NULL);

    // All input data objects have been received, post output data object
    postDataObject(new OutputDataType());
  }
};
```

Multiple flow graph branches are distinguished by the type of the transferred data object. Multiple output or input data types are specified by using the `tv` (short for `type vector`) template, which can take from 1 to 5 arguments. DPS decides which path is taken on the graph by matching the output data types of an operation with the input data types of the following operations. DPS also verifies at compile time that multiple data types are only specified where appropriate, e.g. a leaf operation may only specify one input and one output data type.

In Figure 49, the `Split` operation can output both `DataType1` and `DataType2`, causing the execution of either `ProcessData1` or `ProcessData2` respectively. Such a `Split` operation is declared as follows:

```cpp
class Split : public dps::SplitOperation
  < InputDataType, dps::tv<DataType1,DataType2>, UserThreadType >
```
Routing functions

The role of a routing function is to select a thread within a thread collection on which an operation is to be executed. DPS provides a collection of basic routing functions that can be used for common scenarios:

- **ZeroRoute** – Always returns 0. It is typically used for routing to split and merge operations running within a thread collection containing a single thread that is mapped to a master node.
- **ConstantRoute<value>** – Returns the constant passed as template argument. \( \text{ConstantRoute}<0> \) is identical to ZeroRoute.
- **RoundRobinRoute** – Automatically distributes the data objects in round-robin fashion to all the threads within the target thread collection. It is typically used to achieve a simple distribution on worker threads in embarrassingly simple parallel applications.
- **RandomRoute** – Returns a random index within the target thread collection.
- **NoRoute** – Returns the same index as the previous operation was run on. It is typically used when a data object should stay within the same processing node.
- **LoadBalancedRoute** – Requests the automatic load balancing feature provided by the DPS library.

For more complex data-dependent routing, the developer can provide his own routing functions by deriving from the base class Route.

```cpp
class UserDefinedRoute : public dps::Route<InputDataType>
{
    IDENTIFY(UserDefinedRoute);
    size_t route(InputDataType *in)
    {
        return /* A thread index based on input data object */;
    }
};
```

The Route class takes as a template parameter the type of data object to be routed, which is used as parameter to the route method. The route method returns a thread index within the target thread collection, using only the input data object and the thread collection size. No further information is available, since the routing function is not attached to any particular thread. The number of threads in the collection to which the data object is sent can be obtained by calling the threadCount member function of the Route class.

Flow graph loops

Flow graph loops enable repeated execution of sequences of operations within parallel schedules. A loop construct is composed of a condition and a target operation. When a data object reaches a loop construct in a flow graph, the condition is evaluated. If the condition is true, execution in the flow graph continues at the target operation of the condition, otherwise execution continues on the next operation in the flow graph. The condition is implemented within a class deriving from the base class Condition. Like routing functions, the condition needs to take its decision based only on the content of the incoming data object.

```cpp
class UserCondition : public dps::Condition <InputDataType>
{
    public:
    bool condition(InputDataType *in)
    {
        return /* true or false */;
    }
    IDENTIFY(UserCondition);
};
```

The loop is inserted within a flow graph using a FlowgraphLoop element. The FlowgraphLoop is used in the same fashion as the FlowgraphNode, by inserting it into the flow graph builder with >> operators. The template parameter of the FlowgraphLoop is the condition to use, and its single argument is the target of the loop when the condition is true (in the above example the ProcessData operation).
Figure 50 illustrates a simple flow graph segment containing a loop. This segment can be created with the following source code fragment:

```cpp
// Declare the flow graph nodes (operations)
dps::FlowgraphNode<ProcessData, ProcessRoute> processData(processingThreads);

// Declare the flow graph nodes (loops)
dps::FlowgraphLoop<LoopCondition> loop(processData);

// Add a chain to flow graph
graphBuilder += operation1 >> processData >> loop >> operation2;
```

### Executing parallel schedules

The previous sections described all the components that can be used for constructing thread collections and flow graphs. Once a flow graph has been created, it can be executed as a parallel schedule.

```cpp
InputDataObject *in = new InputDataObject();
OutputDataObject *out = (OutputDataObject*)
    getController()->callSchedule(theGraph, in);
```

The `callSchedule` method starts an instance of a parallel schedule, using the specified flow graph and input data object. When the execution of the schedule is complete, the resulting output data object is returned. It is also possible to invoke parallel schedules asynchronously.

```cpp
InputDataObject *in = new InputDataObject();
dps::ScheduleId scid = getController()->callScheduleAsync(theGraph, in);
// Do something else while the parallel schedule is running
OutputDataObject *out = (OutputDataObject*)
    getController()->waitForSchedule(scid);
```

DPS allows multiple parallel schedules to be executed simultaneously based on the same flow graph and thread collections, as shown in the following example.

```cpp
InputDataObject *in1 = new InputDataObject();
InputDataObject *in2 = new InputDataObject();
dps::ScheduleId scid1, scid2;

// Call schedule twice asynchronously with different input data objects
scid1 = getController()->callScheduleAsync(theGraph, in1);
scid2 = getController()->callScheduleAsync(theGraph, in2);
// Do something else while the parallel schedules are running

// Collect output data objects of both schedules
OutputDataObject *out1 = (OutputDataObject*)
    getController()->waitForSchedule(scid1);
OutputDataObject *out2 = (OutputDataObject*)
    getController()->waitForSchedule(scid2);
```
Object serialization

The data objects that are passed along the flow graph from one operation to another are regular C++ objects. They must be *serializable* in order to be sent over a network connection. Since C++ provides no built-in reflection and serialization capabilities, DPS provides its own mechanisms.

Simple serialization

The simplest mechanism transfers data objects by performing a memory copy. This works for simple data objects without pointers and dynamically allocated memory, and for applications running within homogeneous environment (same platform, same compiler). This mechanism is used by deriving the data object from `dps::SimpleSerial`, and by adding the `IDENTIFY` macro within the object.

```cpp
class SimpleDataObject : public dps::SimpleSerial
{
  IDENTIFY(SimpleDataObject);
  int a;
  int b;
};
```

Automatic Serialization

A more general solution provides the ability to serialize many different data types, including STL containers, arrays and simple pointers. For example, consider the following complex data object:

```cpp
class ComplexDataObject
{
  double a;               // A double
  int c[32];              // Fixed-size array of integers
  std::vector<std::string> strings;            // A vector of strings
  std::map<int, std::set<std::string> > myMap; // A map of sets of strings
  float f;                // A float
};
```

This complex data object is made serializable by deriving from `dps::AutoSerial` and by adding several macros to the data declaration.

```cpp
class ComplexDataObject : public dps::AutoSerial
{
  CLASSDEF(ComplexDataObject) // Declare class name
  MEMBERS                     // Declare class members
    ITEM(double, a)           // A double
    ARRAY(int, c, 32)        // Fixed-size array of integers
    ITEM(std::vector<std::string>, strings)   // A vector of strings
    ITEM(std::map<int, std::set<std::string> >, myMap) // A map of sets of strings
  CLASSEND;

  // Members outside macros will not be serialized and consume no bandwidth
  float f;
};
```

The class must always start with the `CLASSDEF` macro, indicating the name of the type. The functionality of `CLASSDEF` is similar to that of `IDENTIFY`, therefore `IDENTIFY` must not be used. The next element is the `MEMBERS` macro. The list of members is then given with a set of `ITEM` macros, each taking two arguments: the type of the member variable, and the name. These macros simultaneously declare the items. Finally the declaration of the class is completed with the `CLASSEND` macro. In addition to item, DPS also provides the `ARRAY` macro for creating fixed size arrays (in the above example, the array is `int myArray[10]`).

`ITEM` creates public members. Access specifiers such as `private` or `protected` can be added by using the macros `PRIVATEITEM` or `PROTECTEDITEM` (`PUBLICITEM` can also be used and is equivalent to `ITEM`).

Many types can be used in `ITEM` declarations: all the simple types (`int, char, float, etc.`), the STL types (`std::string, std::vector, etc.`), and all serializable types.
If a class inherits from another serializable class, \texttt{BASECLASS} macros must be specified after \texttt{CLASSDEF} to identify the base classes of the data object. Only base classes containing serializable data need to be specified. Since the \texttt{dps::AutoSerial} class contains no such data, it may be omitted.

```cpp
class A : public dps::AutoSerial
{
    
    CLASSDEF(A)
    MEMBERS
       ITEM(double, d);
    CLASSEND;
};

class B : public A
{
    
    CLASSDEF(B)
    BASECLASS(A) // Serializes members of super class
    MEMBERS
       ITEM(A, a) // We may use any serializable object as member
    CLASSEND;
    
    void f(){ std::cout << "this->d = " << d << std::endl; }
};
```

\textbf{Pointers}

DPS also supports objects stored by pointers in classes. The simplest form is a simple pointer to a single object:

```cpp
ITEM(dps::SingleRef<
MyObjectType>, ptr) // MyObjectType *ptr;
```

The \texttt{dps::SingleRef<type>} template provides semantics similar to \texttt{type *}. \texttt{type} must implement the \texttt{dps::ISerializable} interface, e.g. by deriving from \texttt{dps::AutoSerial}. \texttt{dps::SingleRef} takes ownership of the object assigned to it, i.e. it will release the object on its destruction, without incrementing the reference count on assignment. If the object is to be used within other contexts, it is necessary to call \texttt{addRef} on it before assignment in order to ensure that it will not be destroyed when the \texttt{dps::SingleRef} goes out of scope.

DPS also provides variants of \texttt{std::vector} and \texttt{std::map} with pointers in the \texttt{dps::VectorRef} and \texttt{dps::MapRef} classes. These are declared as follows:

```cpp
template<typename RefType> class VectorRef :
    public std::vector<RefType*>;

template<typename KeyType, typename RefType> class MapRef :
    public std::map<KeyType,RefType*>;
```

The ownership semantics for the pointed objects are identical to \texttt{dps::SingleRef}.

\textbf{Simple classes}

The \texttt{dps::AutoSerial} base class adds virtual functions to the class. This is often undesirable for simple helper classes, such as 3D vectors, where the class needs to be kept simple and compact. For instance, consider the following class:

```cpp
struct Vector3
{
    float x, y, z;
};
```

This class has a size of 12 bytes, and no constructors, destructors or virtual functions. It is possible to use such classes efficiently as value types, for example in STL containers. DPS provides a mechanism to make such classes serializable without the memory overhead of the virtual functions in \texttt{dps::AutoSerial}.

```cpp
struct Vector3
{
    SIMPLECLASSDEF(Vector3)
    ITEM(float, x) ITEM(float, y) ITEM(float, z)
};
```
This redefined class is identical to the previous declaration, with only some types and static member functions added. Its size is therefore still 12 bytes, and its usage semantics unchanged. It can be used by value in any complex class derived from `dps::AutoSerial`. Since this class does not provide any of the virtual functions expected in `dps::ISerializable`, it cannot be used by reference in the abovementioned reference types.

**Templates**

DPS also supports serialization on templated classes, by replacing `CLASSDEF` with `TEMPLATEDEFx` (where `x` is the number of template arguments). Consider the following example:

```cpp
template<typename t> class MyTemplate : public dps::AutoSerial
{
    TEMPLATEDEF1(MyTemplate,t)
    MEMBERS
        ITEM(t,templatedMember)
    CLASSEND;
};
```

**Memory blocks**

DPS also provides support for memory blocks of simple types. This is provided by the `dps::Buffer<Type>` class. `Type` can be any simple type, such as `int` or `float`. Since `Buffer` does not call the serializer for every element it contains, it should not be used for complex types (although it will work if the type can be effectively transferred with a simple memory copy). The size of the buffer can be set upon construction, or by calling the resize member function. The size is specified as a number of elements.

```cpp
dps::Buffer<int> buf1(1024);  // Allocate 4096 bytes (1024 ints)
dps::Buffer<int> buf2;
buf2.resize(1024);            // Allocate 4096 bytes (1024 ints)
```

**Logging**

DPS provides a flexible logging mechanism for managing information and debug output. The mechanism uses a hierarchy of logs, where child logs inherit the properties of their parents. A log can simply be created by creating an instance of `dps::Log`.

```cpp
dps::Log myLog("MyLog",NULL);
dps::Log myChildLog("Child",&myLog);
```

The first argument to the constructor is the name of the log, which is used both for display and for configuration. The second argument is the parent log, or `NULL` if this is a top level log. A third optional argument is a Boolean variable indicating whether the log should be used in raw mode (false by default). The fourth argument allows a file name to be specified for the logging output. The output of this log and all its children will be redirected to the file. Output is produced using the `write` method, which behaves like a standard C++ `std::cout`.

```cpp
myLog.write(0) << "Hello world";
myChildLog.write(0) << "Hello again";
```

The single argument to the `write` method is the log level, which specifies how important the message is. By default, only level 0 messages are output. Also, contrary to `std::cout`, the DPS logs will always output a carriage return after the log message. Therefore the two lines above generate the following output:

```
MyLog:0:Hello world
MyLog:Child:0:Hello world
```
If the logs are opened in raw mode, the prefixes with the log names and log levels are omitted. Raw logs are useful if the log content is subsequently postprocessed by another application. In such cases, the log's target will usually be a file:

```cpp
dps::Log results("Results",NULL,false,"results.csv");
results.write(0) << currentIteration << "," << currentParameter;
```

### Logging configuration

The log level, which specifies the threshold at which messages are printed, can be changed at any time by calling `setLogLevel`, and queried by calling `getLogLevel`.

```cpp
myLog.setLogLevel(2);
myLog.write(0) << "Current log level is: " << myLog.getLogLevel();
```

The logs can also be controlled by using the DPS configuration mechanisms. The logging level for a log can be set by using the following options (based on the example above):

```bash
./MyApp -loglevel.myLog 2 -loglevel.myLog.Child 3
```

The logging target can be set in a similar fashion:

```bash
logtarget.myLog=myLog.txt
logtarget.myLog.Child=stdout
```

The special file name `stdout` can be used to redirect log output to the standard output. The settings for a parent are always inherited by all its child logs, unless the child logs specify an override. The options can be specified on the command line as shown above or within the configuration file.

### DPS command line options

The DPS framework itself provides many command line options. The sections below list the options that are directly parsed by the DPS framework. Some components of the framework, such as the thread mapping generators, the connectors or the logging system, are listed in further sections of the reference.

#### General options

- `-cfg` Specifies the configuration file to load for additional options. The default value is `dps.cfg`.
- `-connector` Specifies the connector to use for launching remote applications. The default `LocalConnector` launches additional instances of the application on the local host. The connectors are documented separately.
- `-trace` If set, enables performance tracing. The performance tracing functionality is documented separately.
- `-traceDir` Directory in which the performance trace should be stored. Default is the current directory.
- `-ft` If set, enables fault tolerance within the DPS library.
- `-session` Reserved for internal DPS usage. This option is set on all secondary instances of an application launched by DPS:
- `-dbg` Enables the built-in DPS debugging service. Specifies the port number to listen on, the default being `8888`. The debug service can be connected to with a standard telnet client and provides internal debug information when it is compiled into DPS.
- `-network` Specifies the network layer to use. The default value `TCPNetworkHost` is used for TCP/IP networking. DPS does not currently provide any other network layers.

#### Network options for the TCP network layer

- `-net.port` Specifies the TCP port to listen on. By default, the system chooses an unused port. The port should be identical to the first host used in thread mapping, in order to avoid the creation of an additional application instance. This option should not be used when using the `KernelConnector`, since the kernel will be the network connection point.
- `-net.host` Use this to specify the complete hostname:port for the DPS application. This should be used when the automatic hostname selection of DPS fails, or a specific hostname needs to be used.
By default, the system detects the hostname, and the port can be overridden separately with `-net.port`. If the port is omitted in the parameter, it is automatically selected by the system.

**-net.nodns**

Forbid DNS resolution of hostnames, assuming that all hostnames are of the numerical `xxx.yyy.zzz.www` form. By default, DNS is used to resolve hostnames, unless they start with a number.

**-net.nagle**

Enable Nagle's algorithm on TCP connections. This is disabled by default in order to ensure minimal communications latency.

### TCP network layer: connectors and kernels

While MPI provides its own launching mechanism using a daemon (e.g. `smpd`) and a launcher (e.g. `mpiexec`), applications using raw TCP sockets must provide similar functionality on their own. Connectors are the objects responsible within the DPS framework for creating remote application instances. DPS provides three connectors: `LocalConnector`, `RSHConnector`, and `KernelConnector`. The connector to use can be selected with the `connector` option in the configuration file (or the option `--connector` on the command line).

#### LocalConnector

This connector is provided for testing and debugging purposes. All additional application instances are executed locally. When using the LocalConnector, the mapping strings used should contain only port numbers, for example `:8000 :8001 :8002` if three instances are desired.

By default, an application that is executed from the command line will start listening on a port selected by the operating system. Therefore, when using the above example, four instances of the application will be running after the mapping: the initial instance of the application listening on an arbitrary port, and three instances listening on ports 8000, 8001, and 8002. In order to reuse the initial instance for running the first thread, its port number can be set explicitly to 8000 by specifying the additional command line option `--net.port 8000`. When this option is used, no additional instance of the application is started for port 8000, since the initial instance is already running and listening on this port.

When using the local connector, the following lines would be used in the `dps.cfg` configuration file for a simple compute farm type application:

```
connector=LocalConnector    # Specify the local connector
net.port=8000             # Specify initial port number for first app instance
main=:8000                # One thread in 'main' collection
proc=:8000 :8001 :8002    # Three threads in 'proc' collection
```

#### RSHConnector

This connector is provided for use on clusters that provide `rsh` or `ssh` services for remote login. For every additional application instance required, the connector attempts to run it on the remote host indicated in the mapping string with `rsh`. A typical mapping string would contain a series of host names, for example `lspcluster01 lspcluster02 lspcluster03`. In order to ensure that all hosts use the same port number, it needs to be specified either in the configuration, with `net.port=xxxx`, or explicitly within the mapping string for each host.

By default, the `RSHConnector` executes applications with `/usr/bin/rsh` without specifying a user name. The command to execute can be changed with the `rsh` option, for example `rsh=/bin/ssh` to use `ssh` instead. An additional username can be specified with the `username` option, for example setting `username=someone` produces the remote execution command line `/usr/bin/ssh -l someone myapp` (where `myapp` is the name of the application, filled in automatically by DPS). When using `rsh` or `ssh` it is important to ensure that the remote login does not required any user interaction, for example by asking for a password. For `rsh`, this is typically handled with an `.rhosts` file, and for `ssh` with an `authorized_keys` file. Please examine the `rsh` or `ssh` documentation for more information on configuring these services.

When using the `rsh` connector, the following lines would be used in the `dps.cfg` configuration file for a simple compute farm type application:

```
connector=RSHConnector    # Specify the rsh connector
rsh=/bin/ssh              # Use /bin/ssh as a remote shell
net.port=8000             # Specify port number for all app instances
main=host1                # One thread in 'main' collection
proc=host1 host2 host3    # Three threads in 'proc' collection, on three hosts
```
While \textit{rsh} and \textit{ssh} can take care of most connection scenarios, DPS also provides its own remote execution service in the form of the DPS kernel. The kernel provides similar services to a standard remote shell daemon by allowing the execution of applications, but it provides slightly different handling of port numbers. When using the kernel, it needs to be running on all participating nodes. The kernel is listening on a fixed port, specified by the option \texttt{kernel.port}. The connector will contact the remote kernels and invoke the applications. The thread mapping string should contain only host names, the kernel port number will be used automatically. Using the kernel is typically faster than using \textit{ssh}.

The kernel can be run with the following command line:

\begin{verbatim}
kernel [-kernel.port <kernelport>]
\end{verbatim}

Typically, the kernel port is specified in the configuration file. When using the kernel connector, the following lines would be used in the \texttt{dps.cfg} configuration file for a simple compute farm type application:

\begin{verbatim}
connector=KernelConnector # Specify the kernel connector
kernel.port=5000          # Specify port number for all kernel instances
main=host1                # One thread in 'main' collection
proc=host1 host2 host3    # Three threads in 'proc' collection, on three hosts
\end{verbatim}

The kernel also provides system monitoring functionality compatible with \textit{gkrellm}. This needs to be compiled into the kernel explicitly (see the \texttt{Getting Started} chapter). The monitoring features are activated via the \texttt{-gkrellm} on the command line, with an optional port number on which to listen for \textit{gkrellm} requests (default is 19153).