Computer-Assisted Generation of PVM/C++ Programs Using CAP

Benoit A. Gennart, Joaquín Tárraga Giménez and Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne, EPFL
gigaview@di.epfl.ch

Abstract. Parallelizing an algorithm consists of dividing the computation into a set of sequential operations, assigning the operations to threads, synchronizing the execution of threads, specifying the data transfer requirements between threads and mapping the threads onto processors. With current software technology, writing a parallel program executing the parallelized algorithm involves mixing sequential code with calls to a communication library such as PVM, both for communication and synchronization. This contribution introduces CAP (Computer-Aided Parallelization), a language extension to C++, from which C++/PVM programs are automatically generated. CAP allows to specify (1) the threads in a parallel program, (2) the messages exchanged between threads, and (3) the ordering of sequential operations required to complete a parallel task. All CAP operations (sequential and parallel) have a single input and a single output, and no shared variables. CAP separates completely the computation description from the communication and synchronization specification. From the CAP specification, a MPMD (multiple program multiple data) program is generated that executes on the various processing elements of the parallel machine. This contribution illustrates the features of the CAP parallel programming extension to C++. We demonstrate the expressive power of CAP and the performance of CAP-specified applications.

1 Introduction

Designing a parallel application consists of dividing a computation into sequential *operations*, assigning the operations to threads, synchronizing the execution of the threads, defining the data transfer requirements between the threads, and mapping the threads onto processors. *Implementing* a parallel program corresponding to the design specification involves mixing sequential code with calls to a communication library such as PVM, both for communication and synchronization. While designing the parallel program (i.e. dividing a problem into sequential operations) is an interesting task which can be left to the application programmer, implementing the parallel program is a time-consuming and error-prone effort, which should be automated. Moreover, debugging a parallel program remains difficult.

Portable parallel libraries such as PVM have become widely available, making it much easier to write portable parallel programs. Such libraries allow to write complex parallel programs with a small set of functions for spawning and identifying thread, and for packing, sending, receiving and unpacking messages. The simplicity of a library such as PVM makes it easy to learn and to port to new architectures.

This contribution proposes a coordination language for specifying a parallel program design. The Computer-Aided Parallelization (CAP) framework presented in this contribution is based on decomposing high-level operations such as 2-D and 3-D image reconstruction, database queries, or mathematical computations into a set of sequential suboperations with clearly defined input and output data. The application programmer

Proceedings of EuroPVM-96, LNCS 1156 Münich, Oct. 1996, 259-269

uses the CAP language to specify the scheduling of sequential suboperations required to complete a given parallel operation, and assigns each suboperation to a execution thread. The CAP language is a C++ extension. The CAP preprocessor translates the CAP specification into a set of concurrent programs communicating through communication libraries such as MPI, PVM, and TCP/IP, or communicating through shared memory. The concurrent programs are run on the various processing elements of a parallel architecture. The CAP methodology targets coarse to medium grain parallelism.

Using CAP reduces the effort of implementing parallel programs. CAP specifications produce programs that are deadlock free by construction. CAP programs can be debugged using existing debuggers such as gdb. CAP has been developed in the context of multiprocessor multidisk storage servers. Simple arguments show that careful dataflow control in such architectures is necessary to achieve the best performance. CAP gives the program designer control over the dataflow across threads in a parallel architecture. Both shared- and distributed-memory architectures are supported by CAP.

Previous attempts to build computer-aided parallelization frameworks include the Strand system [11], PCN [5] and the data-parallel fan approach [15] used in SPMD systems. The Strand and PCN approaches are compositional parallel coordination languages with task synchronization mechanisms based on single assignment variables in a global shared name-space. Compositional C++ (CC++) is a parallel programming language based on C++ relying on the experiences made with Strand and PCN [4].

In contrast to Strand, PCN and CC++, CAP synchronization points are not named and are implicitly described by the parallel pipeline constructor. CAP completely separates computations and communications by allowing leaf threads to have only input and output parameters (tokens), without communications. CAP offers more capabilities than SPMD data parallel fans, since participating threads may differ one from another (MPMD). While CAP translates to a static distribution of threads, the amount of data processed by the different threads can vary according to the dynamic run time load characteristics.

This contribution illustrates the features of the CAP parallel programming extension to C++. We demonstrate the expressive power of CAP and the performance of CAP-specified applications. Section 2 explains CAP methodology through an example. Section 3 shows expressive power of CAP and the performance of its PVM implementation.

2 CAP overview

This section introduces the CAP framework : the concepts (section 2.1), the tokens (section 2.2), the thread hierarchy (section 2.3) and the parallel operations (section 2.4). A scalar product example illustrates the concept. Although the example we use is simple, all the concepts introduced can be generalized to arbitrarily complex computations, thanks to the hierarchical and compositional nature of the methodology.

2.1 Concepts

The abstract *parallel server* we consider consists of a set of *threads*. The clients connecting to the server are also modelled as execution threads. Figure 1 represents a parallel server with 3 disks and 3 processors. Each server processor executes two threads,

one for disk accesses (ExtentServer[*]) and one for data processing (Compute-Server[*]). In our model, the client registers to the parallel server interface before starting any computation. The client then connects directly to the server's internal threads (ExtentServer[*] and ComputeServer[*]) to perform parallel computations. The complete server environment consists of the server threads and the client(s) threads. Threads may (but need not) share a common address space.

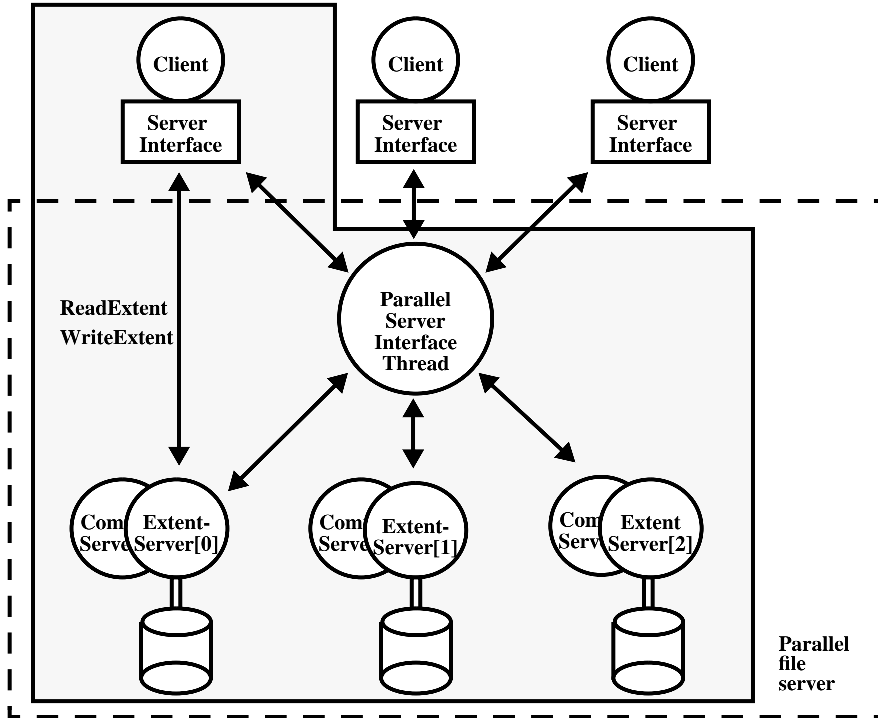

FIGURE 1. Parallel server software architecture

Operations are defined by a single input, a single output, and the computation that generates the output from the input. Input and output of operations are called *tokens*. Communication occurs only when the output token of an operation in transferred to the input of another operation. The basic mechanism for transferring the output token of an operation to the input of another operation is called *redirection*.

Using this terminology, *parallelizing* an operation consists of (a) dividing the operation in suboperations, each of them with one input and one output (the output of a suboperation is an intermediate result of the operation) ; (b) dividing the operation input token into several input subtokens to be fed to the suboperations ; (c) providing the *scheduling* of suboperations required to achieve the result of the original operation ; (d) assigning suboperations to threads in the parallel machine ; (e) merging the result of the suboperations to get the result of the parallelized operation. A schedule indicates the ordering of suboperations required to complete the parallel operation, based on the data dependencies between suboperations. The functions for dividing and merging values are called *partitioning* and *merging* functions respectively.

An operation specified as a schedule of suboperations is called a *parallel* operation. Other operations are called *leaf* operations, are sequential, and are specified in a sequential language such as C++. A leaf operation computes its output based on its input. No communication occurs during a leaf operation. A parallel operation specifies the assignment of suboperations to threads, and the data dependencies between suboperations. When two consecutive operations are assigned to different threads, the tokens are redirected from one thread to the other. Parallel operations are described as communication and synchronization between leaf operations. Threads are organized hierarchically. A set of threads executing on the parallel server is a *composite* thread. The sequential server threads are *leaf* threads. Composite threads execute parallel operations, and leaf threads execute sequential suboperations. Each leaf thread is statically mapped to an OS thread running on a processing element. Each thread is capable of performing a set of leaf operations, and usually performs many operations during the course of its existence. For load balancing purposes, tokens are dynamically directed to threads, and therefore operation execution can be made dependent on instantaneous server load characteristics.

Each parallel application is defined by a set of tokens, representing the objects known to the application, a set of partitioning and merging functions defining the division of tokens into subtokens, a set of sequential operations, and a set of parallel operations. Section 2.2 describes the tokens and partitioning/merging functions for the scalar-product operation. Section 2.3 is a formal description of the threads active in the parallel server. Section 2.4 describes the specification of the parallel scalar-product operation.

2.2 Tokens and partitioning/merging functions

A token consists of application-specific data, and a header indicating which operation must be performed on the data, as well as the context of the operation. Tokens are akin to C++ data structures with a limited set of types : the C++ predefined types, a generic array type, a generic list type, or other token types. This guarantees that the CAP preprocessor knows how to send tokens across any type of network.

```
token VectorPairT {
  int First ;
  int Size ;
  int ComputeServerIndex ;
  ArrayT<double> Left ;
  ArrayT<double> Right ;
} ;
```

```
token ResultT {
  double ScalarProduct ;
} ;
```

PROGRAM 1. Scalar product tokens

In the case of the scalar product operation, there are two token types : the ***VectorPairT*** token, and the ***ResultT*** token. The ***VectorPairT*** token consists of two arrays of floating point values, with the first index and size being ***First*** and ***Size*** respectively. The ***ComputeServerIndex*** filed of the ***VectorPairT*** indicates on which ***ComputeServer*** the scalar product must be computed. The ***ResultT*** token consists of a single floating point value called ***ScalarProduct***. There is one partitioning function called ***SplitVectorPair***, that divides a ***VectorPairT*** in several ***VectorPairT*** slices, and one merging function called ***AddResult***, that accumulates the results of the scalar product operations applied to vector slices.

2.3 Thread hierarchy

When connecting to the parallel server, the client requests from the parallel server the list of server threads, the set of operations each thread can perform, and which input parameters must be provided for each operation. Program 2 is a formal representation of the parallel-server threads. The formal representation is captured by the CAP ***process*** construct. The ***process*** construct consists of the keyword ***process***, the process name, an optional list of subprocesses, and a list of operations. In this example, the parallel server is called ***ParallelServerT***. It contains two ***ExtentServer*** threads, two ***ComputeServer*** threads, and one ***Client*** thread. The set of threads making up the server (***process ParallelServerT***) can itself be seen as a high-level *composite* thread : the thread description is hierarchical. In the thread hierarchy, non-composite threads are referred to as leaf threads. Leaf threads perform sequential operations. Composite threads perform parallel operations. The ***ParallelServerT*** composite thread can perform one operation : the scalar product one a vector pair. The input and output tokens for the ***ScalarProduct*** operation are ***VectorPairT*** and ***ResultT*** tokens.

```
process ParallelServerT {
subprocesses :
  ExtentServerT    ExtentServer[2] ;
  ComputeServerT   ComputeServer[2] ;
  ClientT          Client ;
operations :
  ScalarProduct  in VectorPairT Input
                 out ResultT Output ;
} ;
```

```
process ComputeServerT {
operations:
  SliceScalarProduct
    in VectorPairT Input
    out ResultT Output ;
} ;
```

PROGRAM 2. Parallel server thread hierarchy

Threads run an infinite loop that receives tokens. The thread decodes the CAP header and executes the required operation on the data. The context information contained in the header both the operation that must be started after the current operation, and the address space in which the new operation will run. Communication between threads is asynchronous. Each thread has a mailbox for tokens. A token is kept in the mailbox until the thread is ready to execute it.

2.4 CAP parallel operation

As an example of operation parallelization, consider a scalar product of two vectors. The input token consists of two vectors of real numbers. The output token consists of a single real number, and the computation characterizing the operation is $SP(V_1, V_2) = \sum_{i=1}^{n} V_1(i) \cdot V_2(i)$. A parallelization of this computation would consists of dividing the input vectors in a number of slices sent to the threads in the parallel machine, computing the scalar product on each of the slices, and adding the slices' scalar products to find the vectors' scalar product. In this case, the suboperations are scalar product computations, the partitioning function selects slices in each of the two input vectors, and the merging function adds the slices' scalar product to the vectors' scalar product.

Figure 2 shows the data dependencies and an assignment of suboperations to threads for the scalar product operation, on two vectors with 256 elements each. The ***Client*** process splits the vector pair in slices ; the ***ComputeServer***s carry out the scalar product

operation on the slices ; the ***Client*** adds the intermediate results to compute the slices' scalar product. From this assignment of suboperations to threads, it is easy to figure out the communication requirements for the proposed parallel scalar product algorithm. The slices have to be transferred from the ***Client*** to the ***ComputeServer***s, and the scalar products from the ***ComputeServer***s back to the ***Client***. In the proposed methodology, the application programmer specifies the division of operations into suboperations, the partitioning functions, the scheduling of suboperations (the data dependencies), and the assignment of suboperations to threads. The CAP environment automatically generates the required messages and synchronizations between threads.

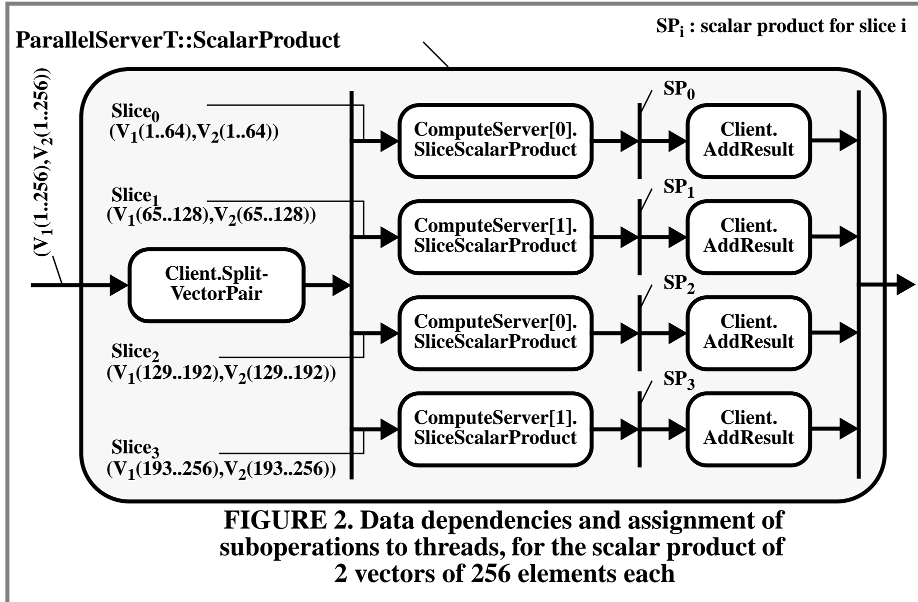

FIGURE 2. Data dependencies and assignment of suboperations to threads, for the scalar product of 2 vectors of 256 elements each

The basis for CAP semantics are directed acyclic graphs (DAGs), which allow to specify the data dependencies for any algorithm. The CAP extension language is rich enough to support the specification of any DAG.

The CAP formal specification of the parallel ***ScalarProduct*** operation (Program 3) consists of a single pipeline expression. The CAP pipeline expression consists syntactically of the keyword pipeline, 4 parameters between parentheses, and a parenthesized body. The pipeline expression divides the input token into 4 slices, using the ***SplitVectorPair*** partitioning function (first pipeline parameter, called the *pipeline-initialization* parameter). The ***VectorPairT*** tokens are sent to the first operation of the pipeline body. In this example, the pipeline body consists of a single CAP operation : ***ComputeServer[Input.ComputeServerIndex].SliceScalarProduct***. The ***ComputeServer*** index is dynamically selected, thanks to the notation ***ComputeServer[Input.ComputeServerIndex]***. ***Input*** refers to the input token of the current operation, in this case a ***VectorPairT*** token. The ***SplitVectorPair*** partitioning function assigns the ***ComputeServerIndex*** field of the input token. The allocation of extents to the various servers is application dependent. This allows to optimize dataflow for each application.

The vector pair slices are redirected to the ***ComputeServer***s (pipeline body : ***Compute-Server[Input.ComputeServerIndex].SliceScalarProduct***). Slice scalar products are redirected to the Client (third pipeline expression parameter, called the *target-thread* parameter), to be merged using the ***AddResult*** merging function (second pipeline expression parameter, called the *pipeline-termination* parameter) into the Result image (fourth pipeline parameter, called the *pipeline-result* parameter).

```
operation ParallelServerT::ScalarProduct
  in VectorPairT Input
  out ResultT Output
{
  pipeline (   SplitVectorPair, AddResult, Client, ImageT Result )
      ( ComputeServer[Input.ComputeServerIndex].SliceScalarProduct );
}

leaf operation ComputeServerT::SliceScalarProduct
  in VectorPairT Input
  out ResultT Output
{ // this is straight C++ code
  int i ;
  Output.ScalarProduct = 0 ;
  for (i = 0 ; i < Input.Size ; i++) {
    Output.ScalarProduct += Input.Left[i] * Input.Right[i] ;
  }
}
```

PROGRAM 3. Parallel ScalarProduct operation

The sequential ***ComputeServerT::SliceScalarProduct*** (bottom of Program 3) consists of a CAP leaf-operation interface, consisting of two keywords (***leaf*** and ***operation***), the operation thread, the operation name, and the operation input and output tokens. The body of a CAP leaf-operation consists of C++ sequential statements, used to compute the output token value based on the input token value.

In the CAP methodology, operations are distinct from threads. Threads are allocated once (usually at the server initialization, or when a new client becomes active), typically remain bound to a single processing element, and execute a large number of operations. Load balancing is achieved by careful allocation of operations to threads. The allocation of operations to threads occurs when setting the ***ComputeServerIndex*** of each ***VectorPairT*** token generated by the ***SplitVectorPair*** partitioning function. The distinction between operations and threads makes for a low overhead in thread management : the creation and destruction of a thread is a rare occurrence. Operations themselves entail little overhead. A typical operation descriptor is small (24 bytes in a PVM implementation), and easy to decode (little mode than a pointer-to-function dereferencing).

```
#include "cap.h"
#include "cap-scalar-product.h"
void main ()

{
  ParallelServerT Server ("ServerName") ;
  VectorPairT* VectorPairP =
    new VectorPairT (... /* initialization parameters */ ) ;
  ResultT* ResultP ;
  call Server.ScalarProduct in VectorPairP out VectorP ;
}
```

PROGRAM 4. Client program

A typical client program is described in Program 4. It opens the server called "Server-Name", and declares two tokens (***VectorPairP***, and ***ResultP***). Using the call instruction, the client runs the parallel ScalarProduct operation. From the client standpoint, the parallel scalar product library looks like almost a sequential library : the only exception is that the client must use the call instruction to call a parallel library operation.

3 Generating parallel PVM programs using CAP

This section gives information on the status of the CAP project in terms of implementation and performance. It gives the current status of the CAP preprocessor implementation (section 3.1), presents an overview of CAP's performance (overhead, and throughput, section 3.2), and lists applications of CAP (section 3.3).

3.1 CAP status

CAP is an extension language to C++. The semantics of CAP parallel expression is based on Directed Acyclic Graphs (DAGs). The CAP language extension supports the specification of any DAG, and the specification of tokens to be exchanged between operations. The translation of DAGs into MPMD programs is explained in [13]. CAP semantics is geared toward the both shared- and distributed-memory machines.

A prototype CAP preprocessor has been implemented, translating mixed CAP/C++ code into straight C++ code calling PVM library routines. The CAP preprocessor generates a program matching the MPMD (Multiple Program Multiple Data) paradigm.

We have tested the preprocessor on a set of examples including an imaging library capable of storing, zooming and rotating 2-D images divided in square tiles, the Jacobi algorithm for the iterative resolution of systems of linear equations, the travelling salesman problem, and a pipelined matrix multiplication.

3.2 CAP performance

To evaluate the CAP overhead, we compare the performance of an actual CAP program with the theoretically achievable performance of the same program. For the theoretical analysis, we measure communication and computation performance, we establish timing diagrams, and derive from the timing diagram the theoretical performance of the parallel program.

Thread management introduces no overhead in CAP. OS threads are initialized when the parallel server is started up, and are statically bound to a specific processing element. The only overhead CAP introduces is due to the token header, representing 24 bytes in our prototype implementation. To evaluate the importance of the token header we compare the token header size to the number of bytes that could be transferred during a typical communication latency (section 3.2.1). We show that on a wide variety of machines, the overhead due to the CAP header is at most 15%, and in most cases below 1%. We then show that the CAP pipeline construct introduces an insignificant amount of overhead, for a wide range of message sizes and computation times.

This section analyses two aspects of CAP performance : the CAP message overhead (section 3.2.1), and the performance of CAP for a simple data fan parallel operation (section 3.2.3). The performance of the CAP construct measured on a network of DEC

workstations connected by an FDDI network. The performance of the FDDI network is experimentally measured in section 3.2.2.

3.2.1 Theoretical CAP token overhead

CAP attaches a header to application-programmer-defined data. In the PVM prototype implementation of the CAP preprocessor, the header size is 24 bytes. In order to evaluate the overhead due to the CAP token header, we compare the header size with the *byte latency cost*, i.e. the number of bytes which could be transmitted during latency time on various parallel architectures and network protocols (Table 1). The numbers in the first two columns of Table 2 are all derived from Dongarra[6]. The third column is a measure of the network quality (throughput divided by latency). The lower the latency and the higher the throughput, the higher the quality. The last column list the byte latency cost of the various communication channels. In the last column, the * entries show the three lowest byte latency cost of all communication channels. The worst-case relative cost of CAP headers for zero-byte synchronization messages on channels having the smallest byte latency cost is thus around 15%. In all other cases, the CAP header overhead will be smaller. In the case where 1KBytes messages are transmitted, the token header represent only 2.4% of the total message size, and in the case of larger messages typical of storage servers (50KB), the token header size is insignificant. Notice also that the byte latency cost is completely independent of the network quality.

Machine/ Network	latency (μs)	throughput (MB/s)	quality (thr./lat.)	lat. byte cost (B)(thr.*lat.)
Convex SPP1200 (sm 1-n)	*2.2*	*92*	*41.82*	*202.4**
Convex SPP1200 (sm m-n)	*11*	*71*	*6.45*	*781*
Cray T3D (sm)	*3*	*128*	*42.67*	*384*
Cray T3D (PVM)	*21*	*27*	*1.29*	*567*
Intel Paragon SUNMOS	*25*	*171*	*6.84*	*4275*
Intel iPSC/860	*65*	*3*	*0.05*	*195**
IBM SP-2	*35*	*35*	*1.00*	*1225*
Meiko CS2 (sm)	*11*	*40*	*3.64*	*440*
nCUBE 2	*154*	*1.7*	*0.01*	*261.8*
NEC Senju-3	*40*	*13*	*0.33*	*520*
SGI	*10*	*64*	*6.40*	*640*
Ethernet	*500*	*0.9*	*0.00*	*450*
FDDI	*900*	*9.7*	*0.01*	*8730*
ATM-100	*900*	*3.5*	*0.00*	*3150*

TABLE 1. Communication performance

3.2.2 Experimental FDDI throughput with PVM protocols

We measure the throughput of the communication channel using a simple ping-pong test, sending of message of a given size back and forth between two workstations. We plot the delay for various sizes and linearize the curve to find latency and throughput figures. The program used to measure the network performance uses the PVM library for communication. The PvmRouteDirect option is turned on, to minimize the number of hops between threads. The delay represents the minimum elapsed time the first message is sent to the time the second message is received, over 20 experiments. Both messages have identical size (ranging from 0 KBytes to 48 KBytes). After linearization of the experimental results, the approximated experimental transfer time t_t is given by the formula :

$$t_t = a + \frac{S}{b} \quad \text{where} \quad \begin{aligned} a &= \text{latency} = 2.2\,\text{msec} \\ b &= \text{throughput} = 4.3\,\text{MB/s} \\ S &= \text{message size} \end{aligned}$$

If we compare with the performance numbers presented in Table 1, our performance measurements are a little under half the best FDDI performance, as measured in [6]. It represents the best performance *we* could achieve on the FDDI network through the PVM communication library.

3.2.3 CAP data fan performance

In this section, we measure the performance of a 'data fan' parallel program, where (1) a master thread sends input data of size S to several slave threads running on separate processors, (2) the threads perform an active loop for duration t_c, and (3) the threads return a result of size S back to the master. The analyzed configuration consists of 4 threads receiving one message each. We perform a theoretical analysis for this program, and compare the result of the theoretical analysis to experimental results.

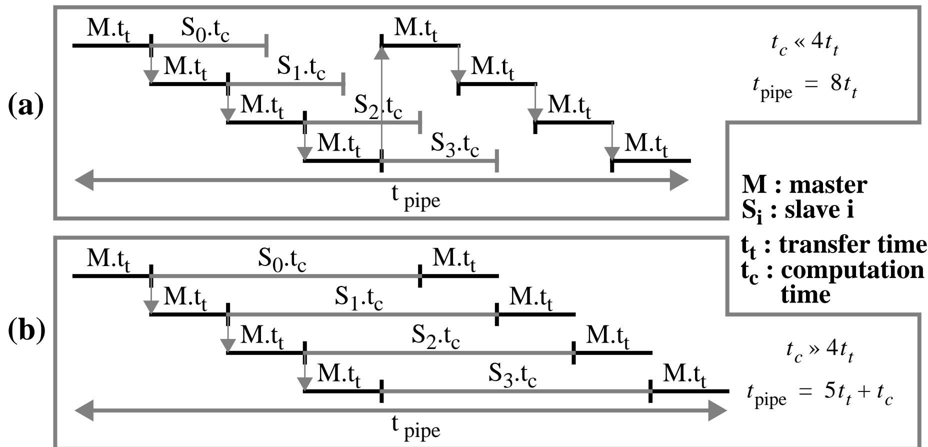

FIGURE 3. Timing diagrams for the pipeline operation

Theoretical analysis. Figure 3 shows timing diagrams for the pipeline operation described in the previous paragraph. Two situations are considered : the *large-message* situation, i.e. the case where the transfer time is larger than the computation time ; and the *small-message* situation, i.e. the case where the transfer time is smaller than the computation time. From the diagrams of Figure 3, we derive two total execution times $t_{datafan}$ for large messages $(t_c \ll 4t_t) \Rightarrow (t_{pipe} = 8t_t)$ (Figure 7(a)) ; and small messages : $(t_c \gg 4t_t) \Rightarrow (t_{pipe} = 5t_t + t_c)$ (Figure 7(b)).

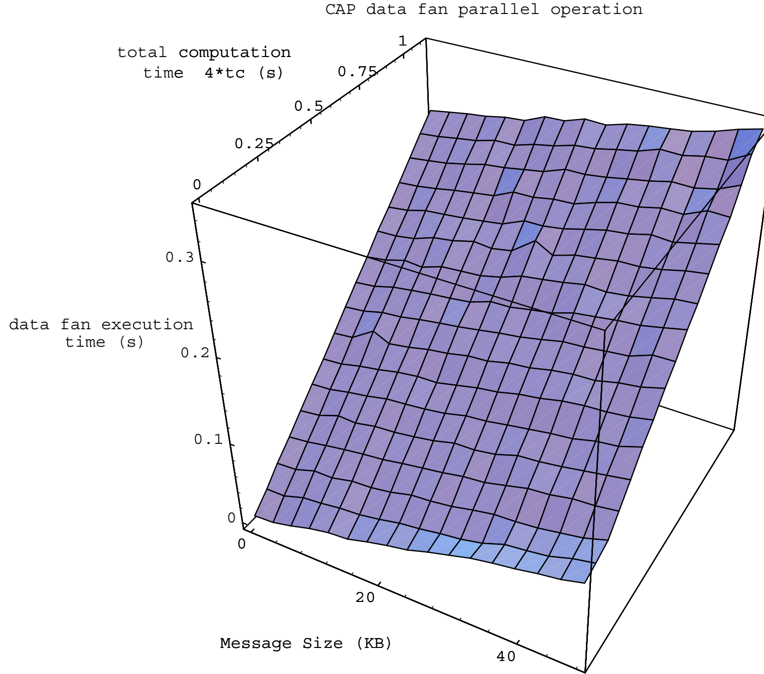

FIGURE 4. Experimentally measured delay

Experimental results confirm these formulas, as shown in Figure 4. Figure 4 plots the delay of the data fan parallel program as a function of the message size S, and the sequential computation time T. The message size S ranges from 0 to 48KB. The combined computation time of all four threads ($4*t_c$) ranges from 0s to 1s. We discuss a few data points in Figure 4. Consider a 48KB message and 0 computation time. The transfer time is $t_t = a + S/b = 13$msec. The computation $t_c = 0$. We are clearly in the 'large message' situation, therefore the total time should be $8t_t$ or 104msec. The experimental measurement in this situation is 89.7msec, slightly better than anticipated by our simple evaluation. In the situation where the message size is 8KB and the computation time per slave is 275msec, we get a communication time $t_t = 4.2$ms. We are in the 'small message' situation. Therefore the total time should be $5t_t + t_c$, or 296msec, for a maximum achievable speed-up of $4t_c / 5t_t + t_c = 3.72$. This is the theoretically achievable speed-up, considering the computation and communication costs described above. The experimental results show a total time of 296msec. These results suggest that for

problems with medium to coarse granularity, CAP does not introduce any noticeable overhead.

The data fan operation performance shown in Figure 4 is application independent. If a given parallel application fits the data fan paradigm, it is sufficient for the application programmer to know the size of the tokens exchanged between operations, as well as the computation time of each of its sequential operations to know the speed-up achievable by a given application. The CAP run-time environment will then deliver close to the theoretically achievable speed-up.

3.3 CAP application examples

CAP is developed in the context of storage servers. Data stored on the server is divided in extents, i.e. data sets with good locality. For example 2-D (3-D) images are divided in 2-D (3-D) tiles. Algorithms have to be parallelized so as to take into account the striping of the data onto the disks. Example of algorithms that benefit from such parallelization are :

- Zooming and rotation on 2-D images.
- Overlay of 2-D maps with different resolutions.
- 3-D image visualization, such as volume rendering by ray tracing [14].

Other more mathematical algorithms would also benefit from parallelization, but we selected spatial problems which require parallelism both at the processing and the storage levels (data striping of files over multiple disks). Such problems involve the use of separate threads for data access (Figure 1, ExtentServer) and data processing (Figure 1, ComputeServer). We have also tested the expressive power of CAP on examples such as the travelling salesman problem and the Jacobi iterative method for solving systems of linear equations.

4 Benefits of the CAP approach

CAP supports pipelining effectively. This is due to the asynchronous nature of communications between threads. Threads leave messages in each other mailboxes, and consult their mailbox when ready to do so. Communication between threads is implicit. The communication is generated automatically from the operation construct scheduling specification.

CAP supports the hierarchical specification of concurrent behavior. An operation specifies the scheduling of suboperations required to achieve an operation. A suboperation can be specified as a scheduling of lower-level operations or as a sequential operation written in C++.

The CAP language constructs are compositional. OS Threads can be grouped in composite threads using the ***process*** construct ; parallel operations are described as scheduling of suboperations ; parallel operations can themselves be instantiated in higher-level parallel operations. Compositionality is essential in storage servers, where multiple users have to access “simultaneously” data on the server. Compositionality ensures that multiple parallel operations can run simultaneously on the server threads, and return data to the appropriate clients. It also ensures that any CAP operation can be called in another CAP operation, making CAP operations completely reusable.

CAP supports shared-memory and distributed-memory architectures. It is capable of sending tokens between threads located in different address spaces. On the other hand, if two threads run in the same address space, the CAP runtime environment will only transfer token references between threads.

The CAP methodology makes for a clean separation between the low level file system (physical blocks storing bytes, messages exchanged between the parallel file system threads at the byte level), and the parallel application-level libraries. The extent servers store contiguous sets of bytes on local disks, and transfer sets of bytes between the various threads of the server. On the other hand, all high-level application-specific functions are created with CAP, which generates the typed high-level messages required by each application, and coordinates the exchange of messages between the various threads in the server to execute parallel operations.

CAP produces applications that are deadlock-free by construction. The specification of a CAP parallel operations is a directed acyclic graph (DAGs), representing the scheduling of suboperations required to achieve the operation. Such a specification is both general and cycle free : all algorithms can be specified so that no earlier step requires the result of a latter step. The lack of cycles ensures deadlock freedom, provided sufficient memory is available. This assertion remains true regardless of the allocation of suboperations to threads. Of course, due to other causes (e.g. infinite loop in a sequential operation, shortage of memory, unreliable communication) a CAP specified parallel program may not terminate, but these are not deadlock situations.

CAP does not entail overhead due to thread management. Threads are initialized at the beginning of the execution of the parallel program. The allocation of threads to architecture components is static and depends on the architecture. During the execution of a CAP operation, the allocation of suboperations to threads is derived from CAP parallel operation specification and is dynamic : thread selection can be a function of the suboperation-instance input-data). The CAP token header for each operation can be made small (24 bytes in a prototype PVM implementation), compared to the typical latency of a communication network. Experiences show (see section 3.2) that the time to transfer the CAP token header is at most 15% of the network latency, and in many case of the order of 1%.

CAP constructs feature a clean separation between parallel and sequential code. The parallel code is concentrated in CAP's process and operation constructs, which represent a small part of an application.

CAP requires a small interface to the communication library. Only a few routines are required by the CAP preprocessor to generate parallel code. These routines are : spawn-new-thread, attach-to-existing-thread, send-message, receive-message, pack-data-structure-into-message, unpack-data-structure-from-message.

CAP supports the automatic generation of application-specific protocols, for exchanging messages between client and server threads. The CAP preprocessor can generate code for transferring a fairly complete set of data structures consisting of a hierarchy of C++ basic types, generic list types, and generic array types.

CAP separates the concepts of computation, communication and execution thread. The communication pattern can be modified easily without changing the computation, by

modifying a CAP parallel operation. The assignment of computation to thread can also be modified easily, again by modifying a CAP parallel operation.

5 Conclusion

We have presented in this contribution a methodology for specifying parallel programs, based on hierarchical directed acyclic graphs. The methodology lets the designer specify a parallel problem decomposition in terms of tokens (C++ structures), data-partitioning functions specified in C++, sequential operations specified in C++, and CAP parallel operations specified as schedulings of sequential suboperations. The methodology allows to generate deadlock free parallel program.

This contribution shows that the overhead introduced by CAP is very low, and that the CAP preprocessor delivers the speed-up predicted by theoretical analyses. We have developed a simple library for parallel imaging, capable of performing zooming and/or rotation on images divided in square tiles stored on multiple disks. We have also applied the CAP methodology to the specification of parallel solutions to the travelling salesman problem and the Jacobi iterative method for solving systems of linear equations.

Future work will aim at demonstrating the performance of more complex parallel applications written in CAP, applying CAP toward distributed systems, and the developing parallel libraries for imaging, video and numerical applications.

References

[1] American National Standard Institute. The programming language Ada reference manual. Lecture Notes in Compute Science 155. Springer-Verlag, 1983.

[2] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy and Ernesto Su. *The Paradigm Compiler for Distributed-Memory Multicomputers*. In IEEE Computer 28(10), October 95, pages 37-47.

[3] Per Brich Hansen. Model programs for computational science : a programming methodology for multicomputers. Concurrency: Practice and Experience, Vol. 5(5), p. 407-423 (August 1993).

[4] K. M. Chandy and C. Kesselman. CC++ : a declarative concurrent object-oriented programming notation. *Research directions in Object Oriented Programming*. MIT Press, 1993.

[5] K. M. Chandy and S. Taylor. *An introduction to parallel programming*. Jones and Bartlett (1992).

[6] Jack Dongarra and Tom Dunigan. *Message-Passing Performance of Various Computers*. University of Tennessee and Knoxville,Tech report 95-299, 1995. URL: http://www.netlib.org/utk/people/JackDongarra.html.

[7] Ian Foster. *Designing and building parallel programs : concepts and tools for parallel software engineering*. Addison-Wesley publishing company. Reading, Massachussets, 1995. ISBN : 0-201-57594-9.

[8] I. Foster and K. M. Chandy. Fortran M : a language for modular parallel programming. In *Journal of Parallel and Distributed Programming*.

[9] Ian Foster and Carl Kesselmann. Language constructs and runtime systems for compositional parallel programming. In *Proc. COMPAR94 - VAPP VI* (B. Buchberger and J. Volkert, Eds.). LCNS 854, Springer-Verlag, p. 5-16, Sep. 1994.

[10] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming : the PCN approach. In *Scientific Programming*, Vol. 1, p. 51-66, 1992.

[11] Ian Foster and Stephen Taylor. *Strand : new concepts in parallel programming*. Prentice Hall, Englewood Cliffs, New Jersey 07632. 1990.

[12] B. A. Gennart and R. D. Hersch. Comparing multimedia storage architectures. In *Proc. Int. Conf. on Multimedia Computing and Systems*, IEEE Press, p. 323-329, Washington 1995.

[13] R. D. Hersch. Parallel storage and retrieval of pixmap images. In *Proceedings of the 12th IEEE Symposium on Mass Storage System*, pages 221-226, Monterey, 1993.

[14] Philippe Lacroute and Marc Levoy. Fast volume rendering using shear-warp factorization of the viewing transformation. In *SIGGRAPH'94 : Computer Graphics Proceedings*, Annual Conference Series, pahes 451-458, Orlando, FL, July 24-29 1994. ACM.

[15] Albert Y. Zomaya. Parallel Computing : paradigm and applications. International Thomson Computer Press, London 1996. URL : http://www.thomson.com/itcp.html