Unified Parallel C
at
the George Washington University

Ludovic Courtès

The George Washington University

4 March, 2002

Abstract

During six months, from the end of July 2001 to the end of January 2002, I worked as a 'research assistant' in the High-Performance Computing Laboratory (HPCL) of the George Washington University (GWU) located in Washington, DC, USA, and led by Dr. Tarek El-Ghazawi. This report presents the Unified Parallel C (UPC) Project and the various tasks that have been assigned to the HPCL and on which I had the opportunity to work during the Fall 2001 semester. UPC is a parallel programming language designed for high-performance computations with supercomputers and is currently still under development.

Keywords: UPC, parallel programming, high-performance computing, compilers.

Contents

0. Acknowledgments

I would like to thank all the people who helped me during this semester in the HPCL: Tarek El-Ghazawi, associate professor at the George Washington University, Sébastien Chauvin, student at the UTBM, for his help getting me started at the beginning of this internship, Frédéric Vroman, student at the UTBM with whom I worked on UPC, Hossam Abdallah, Mohamed Taher, Sinthop Kaewpijit, Preeyapong Samipagdi and Suboh Suboh, my colleagues in the laboratory, for their kindness and help. Thanks also to Gary Funck from Intrepid Technology Inc. and Brian Wibecam from Compaq Corporation for their help with UPC and support of (respectively) the SGI GNU-UPC and Compaq UPC compilers.

1. Introduction

During six months, from the end of July 2001 to the end of January 2002, I worked as a 'research assistant' in the High-Performance Computing Laboratory (HPCL) of the George Washington University. The HPCL works on various projects related to High-Performance Computing. However, during this training-period, I worked only on the Unified Parallel C Project (UPC) and on several tasks related to the development of this parallel programming language. The following section describes with further details the UPC Project itself and also the role of the HPCL in this project.

2. The High-Performance Computing Laboratory at GWU

The High-Performance Computing Laboratory (HPCL) led by Dr. Tarek A. El-Ghazawi belongs to the ECE Department of the School of Engineering of the George Washington University. The institution is described below.

2.1. The George Washington University

The Geoge Washington University (GWU), located in the city of Washington, DC, was founded in 1821. At the very beginning, it consisted of a single building. It now consists of a bunch of building located in the historical district of Foggy Bottom, a mile and a half northwest of the US Capitol Building, and four blocks away from the White House. In addition to this downtown DC campus, GW has a second one, Mount Vernon campus, located in the state of Virginia.

GW is now the largest institution of higher education in the capital. It has nine major schools, offering hundreds of undergraduate, graduate, and professional programs in fields as diverse as forensics, public health and museum education. Among those schools is the School of Engineering and Applied Science (SEAS) which was created in 1852 as a part of the Columbian College.

2.2. The Electrical and Computer Engineering (ECE) Department

The Electrical and Computer Engineering (ECE) department is one of the five departments of the School of Engineering. It was formed on July 1, 1999, after a reorganization of the Department of Electrical Engineering and Computer Science (EECS).

The Department of Electrical and Computer Engineering offers accredited undergraduate programs in Electrical Engineering (alone or with Premedical option) and in Computer Engineering, and graduate degrees at the Masters, Professional, and Doctoral levels in a number of areas of concentration.

The Department today is chaired by Prof. Branimir R. Vojcic, and has 21 full-time faculty and many adjunct faculty. Among the ECE faculty, there are currently several Fellows of the IEEE. Individual ECE faculty members have published many books, and thousands of scholarly papers.

From the ECE website [1].

2.3. The High-Performance Computing Laboratory

The High-Performance Computing Laboratory (HPCL), led by Dr. Tarek El-Ghazawi, now associate professor in the ECE department at GW, was originally hosted at the School of Computational Science of the Geoge Mason University (GMU) in Fairfax County, Virginia. It moved to GWU in late August 2001. The HPCL focuses on high-performance computing, such as cluster computing and consists of two professors and a group of students from different countries (Egypt, Thailand, Palestine, France) working on mainly three projects, one of which is UPC.

Over the years, the HPCL has had the opportunity to work on different projects sponsored by governmental agencies such as the Department of Defense (DoD) and the National Security Agency (NSA) as well as NASA. Beside the UPC Project described below, the HPCL is working on various other projects including a project related to reconfigurable hardware for HPC (called *Lucite*), other projects related to reconfigurable computing, and also one project related to remote sensing and image analysis with NASA. The HPCL owns most of the hardware needed for these projects: a cluster of 8 dual-processor PCs, several reconfigurable boards (FPGA boards), a dozen of workstations including a Compaq Alpha Workstation (actually loaned to Compaq Corporation).

2.4. The UPC Project

The UPC Project aims at developing a new parallel programming language based on C (UPC as a language is described in depth in section 3.1.4 on page 9). There are many participants involved in this project as being part of the UPC Consortium. Participants include governmental agencies such as NSA, IDA, ARSC, CSC, US DoE (Department of Energy) as well as companies like Compaq, Hewlett Packard (HP), Silicon Graphics Inc. (SGI), Cray Inc., IBM, Sun Microsystems, Intrepid Technology Inc., and academics including GWU, MTU, University of California Berkeley.

The UPC Project started more than two years ago. The HPCL, formerly located at GMU, has taken the lead in coordinating the Project from almost the beginning. It has been actively involved in UPC by contributing to a number of major tasks including the definition of the UPC Specifications version 1.0 [8] as well as the development of a UPC Compilers Testing Suite (see section 3.3). The other field in which the HPCL has had an important contribution is benchmarking, with the implementation of the NAS Parallel Benchmark in UPC (discussed in section 3.4), and of a UPC-specific benchmarking suite (section 3.5). Beside this technical involvment, the HPCL has also been coordinating the efforts of the different members of the Consortium and has been, for instance, the official maintainer of the UPC website[1] and in charge of setting up the UPC booth at the SuperComputing 2000 and 2001 conferences (see section 3.2) as well as several UPC meetings including the UPC Workshops that take place at least once a year.

Most of the vendors involved in this project have a team working actively in developing a UPC compiler. Compaq has developed a UPC compiler which is currently the most mature compiler available, based on their own C compiler as well as a runtime environment, both running under Tru64 Unix. Cray Inc. and SGi both have a compiler which is based on the GNU C Compiler (GCC) (see section 3.5.2). Hewlett Packard has also recently released a first version of its UPC compiler and a Sun compiler is still under development.

[1] UPC Homepage: http://upc.gwu.edu/

UPC is not yet a widely used parallel programming language. MPI is still leading parallel programming, mostly because it has been created as a standard a long time ago (version 1.0 dates back to June 1994 [6] whereas version 1.0 of the UPC Specifications [8] dates back only to February 2001) and because many MPI implementations are available on various platforms now, including vendor-optimized implementations. Moreover, bindings for the MPI libraries exist not only in C but also in Fortran (both APIs are defined by the Standard) which is still quite widely used for scientific applications writing. However, due to the significant effort of the vendors involved in UPC, and also to the fact that the Distributed Shared Memory model of UPC (discussed in section 3.1.3) is far easier to program than message passing, UPC should soon have the potential to compete with MPI and maybe become a major parallel programming language in the forthcoming years.

3. The Unified Parallel C

3.1. Introduction to HPC and Parallel programming

3.1.1. Supercomputer Architectures

Research in High-Performance Computing (HPC) aims at designing machines that would be able to perform computations on large amounts of data as fast as possible. Over the years, researchers and companies have contributed to the design of hardware architectures allowing better performance, based on a parallelization of tasks and of data.

Today, there are mostly two types of SuperComputers: cluster of SMPs[1] and the so-called vector machines. Vector machines feature a bunch of specific SIMD[2] processors which are very costly due to the fact that that those processors have a 'custom design'. The most widely used architecture in super-computing is actually clusters of SMPs. Such a machine consists of a bunch of SMP machines connected together through a high-speed network (which typically uses a particular 'light-weight' protocol faster than a regular IP) enabling the nodes of the cluster to exchange data with a very high bandwidth.

Those machines are called NUMA (Non-Uniform Memory Access) machines because even though the global address space is shared among the nodes of the cluster (i.e. the SMP 'machines' of the cluster), accessing data located on a different node is more *costly* (i.e. takes more time) than accessing local data since any access to *remote* data implies a request to the other node(s) over the network – in other words, the cost of an access to the shared memory space depends on the *physical location* of the data accessed. Access to remote data can be, depending on the machine, supported by specific hardware that may be able to perform caching and prefetching of data and ensure data consistency.

As an example, to run UPC programs from the HPC Laboratory, the two following kinds of machines are used:

- SGI Origin 2000: 4 nodes each of which has 4 processors, with hardware support for memory sharing (it is a "Virtual Memory Machine");
- Compaq AlphaServer SC40: 4 nodes each of which has 4 processors (Alpha processors).

[1] Symmetrical Multi Processor machines, machines having several processors on the same mother-board and sharing the same memory address space (i.e. connected to the same bus).

[2] Single Instruction Multiple Data: several data structures can be processed at the same time

Both of them are actual clusters of SMPs and differ mostly in their hardware support for shared memory as well as in the processors used.

3.1.2. Parallel programming

Developers have approached parallel programming in different ways, taking into account the actual architecture of parallel computers. Thus, for example, the leading position of MIMD[1] kind of machines such as clusters versus vector machines (SIMD) has greatly influenced the design of parallel programming. One approach to parallel programming was the design of an API[2] that would give the programmer the facilities that would allow him to have his program running on the different nodes of a cluster and to make them communicate together. The main example for this is MPI (which stands for Message Passing Interface). There are several implementations of MPI available. It provides a set of calls allowing the different processes runnning on different nodes connected together via some kind of a network to communicate. Thus, MPI does not fit well an SMP machine where the whole address space is physically shared. As shown in figure1, only a few communication functions need to be known by the programmer in order to get started with MPI; the complete API (in both Fortran and C) which includes higher-level functions such as the one described in table 3 (page 18) is defined in the *MPI Standards* [6] which can be found on the MPICH website [2]).

However, using such a library makes the task of writing a parallel program tough because the programmer has to take care explicitly of all the communication among the processes. It also makes programs hard to read because each piece of data exchanged results in one or several calls to the communication library. Figure 1 illustrates this major drawback by comparing a C+MPI and a UPC implementation of a simple matrix-multiplication: the UPC program is shorter and more 'straightforward'. Other researchers have been working on parallel languages in which the compiler would handle all the communication among processes, thus allowing the programmer to focus on the algorithm of his parallel program rather than on the underlying communication implied by each memory access.

3.1.3. The Distributed Shared Memory programming paradigm

Parallel languages can be divided in different categories among which the most important are: shared memory programming paradigm and distributed shared memory programming paradigm. These models are named after the type of architecture they try to 'emulate': the shared memory programming paradigm has a representation of the memory which is the same as the one found on SMP machines (the whole memory is shared and can be accessed transparently by any of the processes) and the distributed shared memory programming paradigm is 'mapping' the architecture of NUMA machines such as clusters. It is important to distinguish between the programming model itself and the actual architecture of the machine: any language conforming to one of these models does not necessarily have to be implemented on the kind of machine 'emulated' by the model. It is, however, likely to perform better on the architecture it mimics.

The shared memory programming paradigm is probably the easiest one to understand and to use: From the programmer's point of view, all the data accessed is shared so any part of the memory can be accessed the same way in a transparent way. The compiler and RTS[1] will take care of eventually fetching data from another node of the cluster when a physically remote part

[1] Multiple Instruction, Multiple Data: several data structures can be processed at the same time by several different processing units

[2] Application Programming Interface

```c
#include <mpi.h>
#include <stdio.h>

double a[N/NUM_PROC][P];
double c[N/NUM_PROC][M];
double b[P][M];

int main(int argc, char** argv)
{

  int i,j,l;
  int rank, size;
  double sum;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* Collective arrays initialization for a */
  for(i=0;i<N;i++)
    for(j=0;j<P;j++)
      a[i][j]=(i*NUM_PROC+rank)*j+1;

  if (rank == 0)
  { for(i=0;i<P;i++)
      for(j=0;j<M;j++)
        b[i][j] = i * N +j + 3;
  }

  /* Process 0 broadcasts array b
   * to all other processes
   */
  for (i=0; i<P; i+=MSG_SIZE/M)
    MPI_Bcast(&b[i], MSG_SIZE, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  MPI_Barrier(MPI_COMM_WORLD);


  /* All MPI processes perform matrix
   * multiplication
   */
  for(i=0;i<N/NUM_PROC;i++)
         for (j=0; j<M; j++) {
           sum = 0;
           for(l=0; l< P; l++)
             sum +=a[i][l]*b[l][j];
           c[i][j] = sum;
         }


  MPI_Barrier(MPI_COMM_WORLD);

  /* End of the multiplication */
  MPI_Finalize();
}
```

```c
#include <upc.h>
#include <stdio.h>

shared [P] double a[N][P];
shared [M] double c[N][M];
shared     double b[P][M];

int main()
{
  int i,j,l;
  double sum;

  /* Collective arrays initialization */
  upc_forall(i=0; i<N; i++; &a[i][0])
    for(j=0;j<P;j++)
      a[i][j] = i * j + 1;

  for(i=0; i<P; i++)
    upc_forall(j=0; j<M; j++; &b[i][j])
      b[i][j] = i * N +j + 3;

  upc_barrier;

  /* All threads  perform matrix multiplication */
  upc_forall(i=0;i<N;i++;&a[i][0])
  { for (j=0; j<M; j++) {
      sum=0;
      for(l=0; l< P; l++)
        sum +=a[i][l]*b[l][j];
      c[i][j] = sum;
    }
  }

  upc_barrier;

  /* End of matrix-multiplication */
  return 0;
}
```

Figure 1. C+MPI versus UPC implementation of a matrix-multiplication

C Function Name	*Description*
`MPI_Comm_rank()`	Returns the rank (i.e. the process number or process ID) of the calling process.
`MPI_Send()`	Sends data to all the processes of a group.
`MPI_Recv()`	Receives data from one given process.
`MPI_Bcast()`	Broadcasts data to all the processes of a group.
`MPI_Barrier()`	Synchronization barrier for all the MPI processes of a group.

Table 1. Main MPI point-to-point communication functions

of this 'virtually shared memory' is accessed. OpenMP is an example of a language implementing this paradigm. The shared memory programming paradigm offers a very simple abstraction layer from the architecture of the machine. However, this abstraction prevents the programmer from having any control on the data accessed. In particular, the main problem is that the pro-

[1]Run-Time System, or Run-Time Environment (RTE)

grammer doesn't know where the data is physically located; therefore, a parallel program in a shared memory paradigm cannot be optimized in order to reduce costly remote accesses.

The Distributed Shared Memory (DSM) programming model addresses this problem without removing the ease of access to shared memory: there is also a shared address space but the programmer can know where the data he wants to access is physically located. Thus, *data locality* can be exploited in order to improve the performances of a program. When writing a program in a language conforming to the DSM model, the programmer can make sure that each process of the program will access as much as possible the data which is physically local (i.e. located on the same node or processor).

3.1.4. UPC as a DSM Language

UPC is a distributed shared memory parallel programming language, originally designed by William W. Carlson et al., and version 1 of the UPC Language Specifications was released in February 2001 [8]. UPC builds on the experience of its various predecessors, also DSM languages keeping a syntax close to the C syntax, such as AC [7] and Split-C. Being such, it offers a particular view of the address space to the programmer, and offers various facilities that allow to take advantage of data locality in a very straight-forward way. The memory is divided between the shared space and the private space. Each UPC thread[1] has its own private space and can also have *affinity* to part of the shared memory. When a thread has affinity to a block of shared memory, this means that this block is *physically local* to the thread.

To declare a shared variable in UPC, the only thing to do is to add the **`shared`** keyword before the declaration of the variable. Thus,

```
shared int a[100];
```

declares an array of a hundred integers distributed among all the threads in a round-robin fashion. Therefore, element `a[i]` has affinity with thread `(i % THREADS)`, `THREADS` being the UPC variable which gives the total number of threads of a program. As described in the UPC Specifications1.0, the **`shared`** keyword is added to the C standard type qualifiers such as **`const`**, **`static`**, **`volatile`**, and subsequently follows the same syntax as these keywords.

In addition, the **`shared`** keyword can be followed by a so-called *layout qualifier* which defines how the elements of the array should be distributed. This layout qualifier consists of a *block size* or *blocking factor* enclosed in square brackets. For instance, if this block size is set to 3, each thread ends up having affinity with blocks of 3 contiguous elements of the array. The declaration for such a distribution would look like this:

```
shared[3] int a[100];
```

Also, the declaration

```
shared int a[100];
```

is equivalent to

```
shared[1] int a[100];
```

The programmer can also have a whole array having affinity to thread 0 only using the *indefinite blocking factor*:

```
shared[] int a[100];
```

[1] A UPC thread is equivalent to what most other parallel programming languages refer to as a process.

```
int i;  /* private integers replicated in every thread */

shared int a[5];    /* shared array */
shared[2] int b[9];

shared int* ptos;  /* private pointer to shared space */
shared int *shared stos; /* shared pointer to shared space */
```

The declarations above lead to the following layout (with 3 threads):

Shared memory space		
Thread 0	Thread 1	Thread 2
a[0]	a[1]	a[2]
a[3]	a[4]	
b[0]	b[2]	b[4]
b[1]	b[3]	b[5]
b[6]	b[8]	
b[7]		
stos		
Private memory space		
i	i	i
ptos	ptos	ptos

Figure 2. Different types of data layout in UPC

Figure 2 gives several examples of declarations and the resulting data layout. Section 3.1.5 describes in further details the main features of the UPC language.

3.1.5. Summary of some UPC features

Dynamic memory allocation

UPC offers in its standard library a set of functions dealing with shared memory allocation. Here are those functions:

- `shared void* upc_all_alloc(B, BS);`

  allocates `B × BS` bytes in the shared memory space, distributed in a round-robin fashion among the threads by blocks of size *BS* bytes; This is a collective function and should be called by all the threads; all of them get a copy of the pointer to the newly allocated chunk of memory;

- `shared void* upc_global_alloc(B, BS);`

  works as the previous one, except that this is not a collective function; therefore, only the calling thread gets a pointer to the allocated memory;

- `shared[] void* upc_local_alloc(B, BS);`

  allocates `B × BS` bytes in the local pool of the shared memory space; Thus, only the calling thread has affinity to the newly allocated piece of memory, hence the indefinite blocking factor of the returned pointer;

- `void upc_free(shared void* p);`

  frees the dynamically allocated memory pointed to by `p`.

Workload distribution

As described above, the DSM model relies on the fact that each process (each UPC thread) should access as much as possible only the data that is local to it (the data that is has affinity with). UPC has some facilities, namely the **`upc_forall`**`(;;;)` statement and the `upc_threadof()` primitive, that help dealing with affinity.

```
size_t upc_threadof(shared void* p);
```

which takes as a parameter a shared pointer returns the number of the UPC thread that has affinity to the data pointed to. The **`upc_forall`**`(;;;)` statement is equivalent to the standard **`for`**`(;;)` statement, except that it adds a fourth field called the *affinity field*. This fourth argument tells which thread execute the iterations, and can be either a scalar (i.e. the number of the thread that will execute the iteration) or a shared pointer. In the latter case, the iteration of the loop will be executed only by the thread which has affinity to the data pointed to. Figure 4 shows four equivalents loops in UPC, some of them using the **`upc_forall`** statement, others using a simple **`for`** statement.

Synchronization mechanisms

UPC offers the main synchronization mechanisms that one would expect from a parallel programming language, especially when data is shared. The main one is the so-called *barrier* which corresponds to the **`upc_barrier`** statement. When one thread reach a barrier at some point, its execution is suspended until all the other threads have reached the same barrier.

Synchronization can also be performed using a *split-phase* barrier which consists of the **`upc_notify`** and **`upc_wait`** statements: the **`upc_notify`** statement is not blocking and should be placed at the critical point of the program that requires synchronization among the threads; once they have reached the **`upc_wait`** statement, all the threads will wait until each other thread has reached the previous **`upc_notify`** statement. In between those two statements, in order to reduce the latency when reaching the **`upc_wait`**, each thread could for instance make some local computations that do not depend on other threads status. Figure 3 illustrates this example.

```
{
        /* computations depending on shared data */
       ...
        /* local computations */
       upc_barrier;
}
```

```
{
        /* computations depending on shared data */
       upc_notify;
        /* local computations */
       upc_wait;
}
```

Figure 3. The split-phase barrier compared to the barrier

Locks

Beside these synchronization mechanisms, the programmer can use *locks*, that is a mechanism close to the one of semaphores. `upc_lock_t` is the type of any UPC lock variable. UPC allows to create such locks either statically by declaring a `upc_lock_t` variable or dynamically using

```
shared int a[SIZE];
...
{ ...
  upc_forall(i=0; i<SIZE; i++; &a[i])
  {  <iteration>  }
}
```

```
shared int a[SIZE];
...
{ ...
  upc_forall(i=0; i<SIZE; i++; i)
  {  <iteration>  }
}
```

```
shared int a[SIZE];
...
{ ...
  for(i=0; i<SIZE; i++)
  { if (MYTHREAD == upc_threadof(&a[i]))
    {  <iteration>  }
  }
}
```

```
shared int a[SIZE];
...
{ ...
  for(i=0; i<SIZE; i++)
  { if (MYTHREAD == i)
    {  <iteration>  }
  }
}
```

Figure 4. The **upc_forall** statement and upc_threadof() primitive: four equivalent loops using them

the upc_lock_[global|all]_alloc() functions. The functions related to locks handling are the following:

- upc_lock_t* upc_lock_global_alloc()

  should be called by only one thread; returns a pointer to the new allocated (and initialized) thread;

- upc_lock_t* upc_lock_all_alloc()

  same as above except that this one is a collective call (called by all threads);

- **void** upc_lock(upc_lock_t* l)

  the thread that calls this function waits until the lock pointed to by l is freed and then acquires it and locks it;

- **void** upc_lock_attempt(upc_lock_t* l)

  same as above except that this is a non-blocking call that returns 1 on success;

- **void** upc_unlock(upc_lock_t* l)

  unlocks the UPC lock pointed to by l.

Shared String Handling

Several functions equivalent to the standard C string handling functions are available in UPC for shared strings:

- **void** upc_memcpy(**shared void*** dst, **shared void*** src, **int** nbytes)

  copies nbytes from src to dst (shared-to-shared copy);

- **void** upc_memget(**void*** dst, **shared void*** src, **int** nbytes)

  copies nbytes from src to dst (shared-to-private copy);

- **void** upc_memput(**shared void*** dst, **void*** src, **int** nbytes)

  copies nbytes from src to dst (private-to-shared copy);

For further description of the UPC features, the reader can refer to the UPC Specifications [8] or to the *UPC Quick Reference Card* both available on the UPC Website.

3.2. The SuperComputing 2001 Conference

The HPCL people involved in the UPC Project attended the SuperComputing 2001 Conference which took place in Denver, Colorado, from November 10th to November 16th (see http://www.sc2001.org/). The conference receives each year HPC vendors along with academic and governmental HPC research institutes. Tutorials and lectures along with research papers presentations are given during two days. Booths for vendors and researchers are installed in an exhibition hall.

For the second time, the UPC Project had its own booth there. Various participants involved in the Project, including sudents from the HPC Laboratory of the GWU, were in charge of presenting the Project and the language and its features to visitors. Beside preparing the booth, the HPC Laboratory was also running a UPC demo based on Sébastien Chauvin's demo from last year. This demo consisted of a graphical user interface (GUI) written in Tcl/Tk along with UPC implementations of the N-Queens problem as well as the Sobel Edge-Detection algorithm. The GUI allowed the user to view a graphical demonstration of the algorithm chosen and to run the actual UPC program in order to get its performance curves.

The work on the new demo mainly consisted in merging the N-Queens and Sobel edge-detection GUIs in order to have something more consistent and more convenient when changes in the code are required. Also, the new version of the demonstration also had an "MPI Run" button beside the "UPC Run" button, so that people could compare the performances of the MPI and UPC versions of these two programs.

3.3. UPC Compilers Testing Suite

During his Fall 2000 internship, Sébastien Chauvin developed a UPC Compilers Testing Suite. The purpose of this testing suite is to help compiler writers in testing whether their compiler is consistent or not with the UPC Specifications [8]. To achieve this goal, a set of programs were written in order to test the conformity of a given compiler with different points defined by the specifications of the language.

During the past semester, compiler writers have been giving some feedback regarding the testing suite. Some test cases implementations were considered incorrect or inconsistent and had to be fixed. In addition to that, a small configuration script that would automatically recognize which platform is being tested was written. Currently, mostly two platforms are supported (SGI Origin with GNU-UPC compiler and Compaq TruCluster with Compaq UPC compiler), so the script is still sufficient to handle the differences between those architectures. However, a real configure script automatically generated by an appropriate tool such as GNU Autoconf may be more appropriate in this situation, at least for future releases.

3.4. NAS Parallel Benchmark in UPC

3.4.1. Implementation

Several benchmarks are available for parallel machines. Among them, one of the most widely used is the NAS Parallel Benchmark which was designed by the NAS department of NASA. The NPB consists of a set of programs (the so-called 'kernels' and the much larger applications) designed to evaluate the performance of parallel machines (including their software

Kernel	*Description*
EP	Embarassingly Parallel algorithm: little or no communication among processes
FT	3-D Fast-Fourrier Transform: quite a lot of communication
IS	Integer Sorting (bucket sort)
MG	3-D scalar Poisson equation with MultiGrid method
Application	*Description*
LU	Solves a block of LU equations
SP	Pentadiagonal solver (Navier-Stokes equation)
BT	Block tridiagonal solver (Navier-Stokes equation)

Table 2. NPB 2.3 kernels and applications

facilities). Each one of those kernels and application is based on a well known mathematical problem whose implementation emphasizes one particular aspect of parallel processing whose performance is to be evaluated. Table 2 shows a list of the kernels and applications of the NPB as of version 2.3.

The NPB developers implemented two versions of the NPB: one sequential version written in C called NPB-serial and one parallel version written mostly in Fortran using the MPI library which is the one most vendors use to measure the performance of their parallel machines. A few students of the HPC Laboratory started last year to rewrite these programs in UPC. The aim was to be able to compare, on a given platform, the performance of the MPI version of the NPB versus the UPC version. Considering the fact that UPC implementations are not yet mature and still lack significant optimizations, porting the MPI code would at the same time allow to identify exactly where the UPC code performs worse and what kind of optimizations would be necessary to make it perform better.

By the end of December, we had four kernels running. In most cases, they do not perform as well as the MPI version, or at least for small problem sizes. However, the main optimizations required have been identified and implemented and we have been able to improve significantly the performance of those kernels. The UPC Benchmark (UPC_Bench) as described in section 3.5 addresses the main optimization issues with current UPC implementations. Another important improvement is the implementation of a collection 'reduction' function which is described in section 3.6 (page 17).

3.4.2. Performance Evaluation

To evaluate the performance of a parallel application, people usually consider two criterias. The first one is, of course, the execution time of the program, but the most useful information about performance is the so-called *speedup* of the application. The speedup of a parallel program running N processes (or N UPC threads) is defined as follows:

$$speedup = \frac{execution\ time\ with\ 1\ thread}{execution\ time\ with\ N\ threads}$$

The ideal speedup for a parallel application would be the curve defined by $speedup = N$, N being the number of processors used to run the program (i.e. the number of processes also). A parallel program whose speedup curve is close to the ideal one is said to *scale*. However, such a speedup is rarely achieved with actual parallel implementation. Most of the time, overhead due to communication among processes makes the speedup far worse than the ideal one. This is especially the case for the Fast-Fourrier Transform kernel (FT) of the NPB whose performance curves are shown in appendix A on page 22.

In order to be able to quickly output execution time and speedup curves of those kernels similar to the one shown in appendix A, a set of tools was written, consisting basically of a *Makefile*, a Scheme script that would compute a list of speedups corresponding given a list of execution times consisting of a basic text file showing the number of UPC threads in a first column and the corresponding execution time in a second column, along with a Lout template. Typing the following:

```
$ make KERNEL=ft CLASS=A
```

would directly output a PostScript file with both curves using the Lout Document Formatting System[1]. This set of tools has further been adapted to be used with the UPC Benchmark (see section 3.5).

3.5. The UPC Benchmark on the SGI Origin 2000

3.5.1. Presentation of the UPC Benchmark

The UPC Benchmark was originally developed in Fall 2001 by Sébastien Chauvin and Tarek El-Ghazawi. This small benchmark aims at evaluating the performance of the available UPC implementations. It consists of three applications which are a Sobel edge detection program, a matrix-multiplication program and the implementation of the N-Queens problem, along with a synthetic benchmark which measures the performance of a number of UPC operations related to shared memory accesses. Sébastien Chauvin and Tarek El-Ghazawi published in 2001 a research paper for the International Conference on Parallel Processing (ICPP) in 2001, analysing the results of the UPC_Bench on the Compaq platform [5].

The N-Queens problem does not require any communication among the threads: each UPC thread looks for possible combination in a tree-fashion, without having to shared any data with the other threads. Subsequently, the N-Queens program should *scale* perfectly, which means that doubling the number of thread should result in dividing by two the execution time. Such an algorithm is also called "embarassingly parallel" (see the NPB in section 3.4). On the contrary, the Sobel edge-detection algorithm requires some communication among threads and the matrix-multiplication program is even more demanding.

For those two last applications, the UPC_Bench aims at showing what performance can be gained by rewriting the critical parts of the UPC code with many shared references with specific 'hand optimizations'. Those optimizations are mostly of two types:

- casting local shared references to private references (figure 5 illustrates this optimization);
- prefetching remote shared data into the private space of a thread prior to any access to it.

On both the Compaq and the SGI platform, those hand-tunings turned out to allow significant gains in performances. This reveals deffi ciencies in the implementations of compilers and run-time environment. For instance, one could expect from the compiler to be able to automatically 'recognize' shared references to local shared data and consider them as simple private references, but as of version 1.7 of the Compaq UPC compiler and version 1.8 of the GNU-UPC SGI compiler, this optimization is still not available. This kind of 'misbehaviour' of the compiler is likely to be addressed by developers in the near future.

---

[1]The Lout Document Formatting System is available at ftp://ftp.it.usyd.edu.au/jeff/lout under the terms of the GNU General Public License.

```
shared int* sp;   /* shared pointer */
int* pp;          /* private pointer */
...

/* make sure that MYTHREAD has affinity to the data pointed to by sp */
if (MYTHREAD==upc_threadof(sp))
{
  pp = (int*)sp;
  /* the local shared data pointed to by sp is now available through
   * the private pointer pp.
   */
  pp[1] = 2;
  ...
}
```

Figure 5. Casting of a local shared reference to a private reference

Appendix B shows typical gains obtained with those hand-optimization with the matrix-multiplication application on an SGI Origin 2000 machine, compared also to the performances of the same program written in C using MPI (page 23). Those curves shows that significant performance improvement can be gained from a pointer-casting type of optimization. Data prefetching, on the other hand, can slightly improve performances when large amounts of remotely-shared data are being accessed.

3.5.2. Implementation on the SGI Origin

The first stable version of the GCC-based UPC compiler for the SGI Origin family[1] was released in November 2001 and we tried to have this benchmark running as soon as possible on this platform. However, as of version 1.8, based on GCC 2.95.2, the GNU-UPC compiler still has some limitations. Among them, the most significant is the fact that the compiler in its current version does not support blocking factor. If a program declares a shared array with a blocking-factor different from one, the compiler would just not compile it. As a consequence, significant changes had to be made to the code of the Sobel and matrix multiplication applications.

Those modifications had to be done in such a way that the data distribution among the threads would be the same as the one expected with the correct blocking factor. Changing the blocking factor of a shared array in UPC does not necessarily change the physical data layout but at least, it does change the way the array is *traversed*. Subsequently, in the Sobel edge-detection program, in order to traverse the lines of the images in the right way, array indices had to be modified in order to have each thread accessing the right rows of the images, that is the lines it has affinity with. Typically, the two array declarations shown in figure 6 result in the same data layout: each thread 'holds' `(IMGSIZE/THREADS)` elements. However, in the original code, thread `#i` has affinity to elements `orig[i*(IMGSIZE/THREADS)]` to `orig[(i+1)*(IMGSIZE/THREADS) - 1]`, whereas in the second implementation (with a default blocking factor of 1), it has affinity to elements `orig[i+k*THREADS]`, with $0 \leq$ `k` $<$ `IMGSIZE / THREADS`.

[1] The official website of the GNU-UPC compiler is located at http://upc.gwu.edu/software/gnu-upc.html

Original Code

```
typedef struct {
  BYTET r[IMGSIZE];
} RowOfBytes;

shared[IMGSIZE/THREADS]
  RowOfBytes orig[IMGSIZE], edge[IMGSIZE];

/* Next and previous rows index */
#define NEXT_ROW(i) ((i)+1)
#define PREV_ROW(i) ((i)-1)
```

Modified Code

```
typedef struct {
  BYTET r[IMGSIZE];
} RowOfBytes;

shared RowOfBytes orig[IMGSIZE], edge[IMGSIZE];

/* Next and previous rows index */
#define NEXT_ROW(i) \
  ((((i)+THREADS)%IMGSIZE)+(((i)+THREADS)/IMGSIZE))
#define PREV_ROW_(i) \
  ((((i)-THREADS)%IMGSIZE)+(((i)-THREADS)/IMGSIZE))
#define PREV_ROW(i) \
  ((i<THREADS)?(PREV_ROW_(i+IMGSIZE)-1): \
  (PREV_ROW_(i)))
```

Figure 6. Macros used to access rows of the source and edge-detected images in the Sobel program: original code and code used with GNU-UPC 1.8

Figure 6 shows macros that were written to choose choose the right way of computing rows indices, depending on whether the program is compiled with this version of the GNU-UPC compiler or not, in order to ease the portability of the program; the right-hand-side code is the one used with GNU-UPC.

3.6. Collective Reduction Function

As we were developping the UPC kernels of the NAS Parallel Benchmark and comparing their performances versus that of the Fortran+MPI versions, we noticed that the performances of the UPC kernels were worse than those of the MPI programs. Beside the fact that current UPC compilers do not support the optimizations described in section 3.5.1, it appeared clearly that the lack of collective operations in UPC was one of the main reason for this result.

MPI defines a set of collective operations, that is a set of collective communication functions (i.e. functions that should be called by all the members of an MPI process group) designed to perform collective operations that requires data from all the processes in an optimized way. Some of those functions that are described in the MPI Standards 1.1 [6] are shown in table 3. It would not make much sense to have all these functions available in the standard UPC library since we are assuming that compilers and run-time environment should be able to automatically optimize remote memory accesses using caching/prefetching methods – in other words, it is up to the UPC compilers and RTE to do the optimization that one would do 'by hand' when programming with MPI.

However, among those collective operations, an 'all reduce' function turned out to be the most important one to implement in UPC to be able to compete with MPI programs. Such a function is required also in UPC because there is no compiler or RTE optimization that could possibly be efficiently substituted to this kind of function. As a matter of fact, most kernels take advantage of the `MPI_Allreduce` function, hence the need for a UPC equivalent function. A simple version of such a function that would suit our needs for the kernels was first written and

Name	*C function name*	*Description*
Barrier Synchronization	`MPI_Barrier()`	Barrier synchronization across all group members
Broadcast	`MPI_Bcast()`	Broadcasts an array from one process to all other processes of a group
Gather	`MPI_Gather()` `MPI_Allgather()`	Gathers data from all group member processes to one process (respectively to all processes)
Scatter	`MPI_Scatter()`	The opposite: dispatches data from one member to all members of a group of processes
All-to-all scatter/gather	`MPI_Alltoall()`	Same as gather and scatter except that data is exchanged from all processes to all processes of a group
Global Reduction Operations	`MPI_Allreduce()`	Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member

Table 3. Main MPI collective communication functions

then extended so that it could be included in a library, based on Libby Wiebel's and Sébastien Chauvin's proposals for UPC collective operations ([4], [3]).

In order to get lowest execution times, this function performs collective operations among threads in a tree fashion: for the first step, half of the UPC threads will perform the operation for their operand and the one of the next thread and keep this as a temporary result; for the next step, one quarter of the threads will compute a temporary result in the same way, and so forth, until we get one final result. The complexity of such an algorithm is $\log_2(THREADS)$. Figure 7 illustrates it in the case of the 4-thread UPC program.

A different prototype than the one proposed before was chosen. The function is available for data of type `int`, `long`, `float` and `double`, and the prototype for the integer version of the function is the following;

```
shared[] int* upc_all_reduce_int
        (int* myoperand, size_t opsize, upc_operator op,
         int (*f)(int,int), size_t ret_threadof)
```

This function computes the operation specified by `op` (which can be one of `UPC_ADD` for an addition, `UPC_SUB`, `UPC_FUNC` for a user defined function pointed to by `f`, and so on). The result of this operation is returned as a pointer to shared data that has affinity to the thread specified by the `ret_threadof` parameter. Therefore, every thread will get a pointer to the result. This is a collective function so each thread should call it with parameter `myoperand` pointing to an array of `opsize*sizeof(int)` elements. The space needed for the result is allocated inside the function by thread `ret_threadof` calling `upc_local_alloc()`.

A performance evaluation of the `upc_all_reduce` function is available in appendix C on page 24. The graphs show a comparison between several method implementing an addition of elements distributed among all threads, including the `upc_all_reduce` tree-fashion implementation. As expected, `upc_all_reduce` is much faster than any of the two other 'straightforward' implementations tested.

Writing a collective operations library will be one of the main goals of UPC soon. It is very likely that vendor-specific implementations of such functions (which would not necessarily be written in UPC but rather in a low-level language such as C) could perform better than this UPC implementation, so long as it can take advantage of the vendor's hardware and software specific features. This is exactly what happens with MPI: even though there is at least one Free and portable implementation (MPICH), most supercomputing vendors have developed their own implementation on MPI that is specifically optimized for their platform. However,

The following diagram explains the tree-fashion algorithm used by the `upc_all_reduce()` function when adding `THREADS` elements (with `THREADS==4`). *v0* to *v3* are private arrays located on threads 0 to 3; the resulting array contains the addition element by element of those 4 arrays.

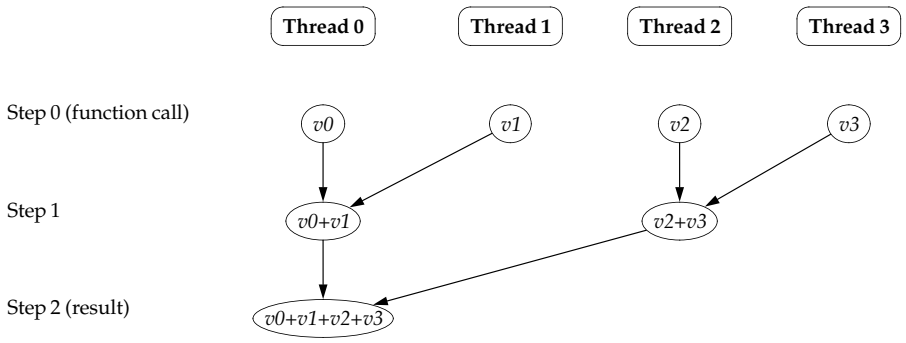


Figure 7. Adding elements in a tree-fashion (with 4 threads)

this implementation of a `upc_all_reduce()` function is UPC already allows to get good performances and vendor-specifi cimplementation would just improve it a bit more.

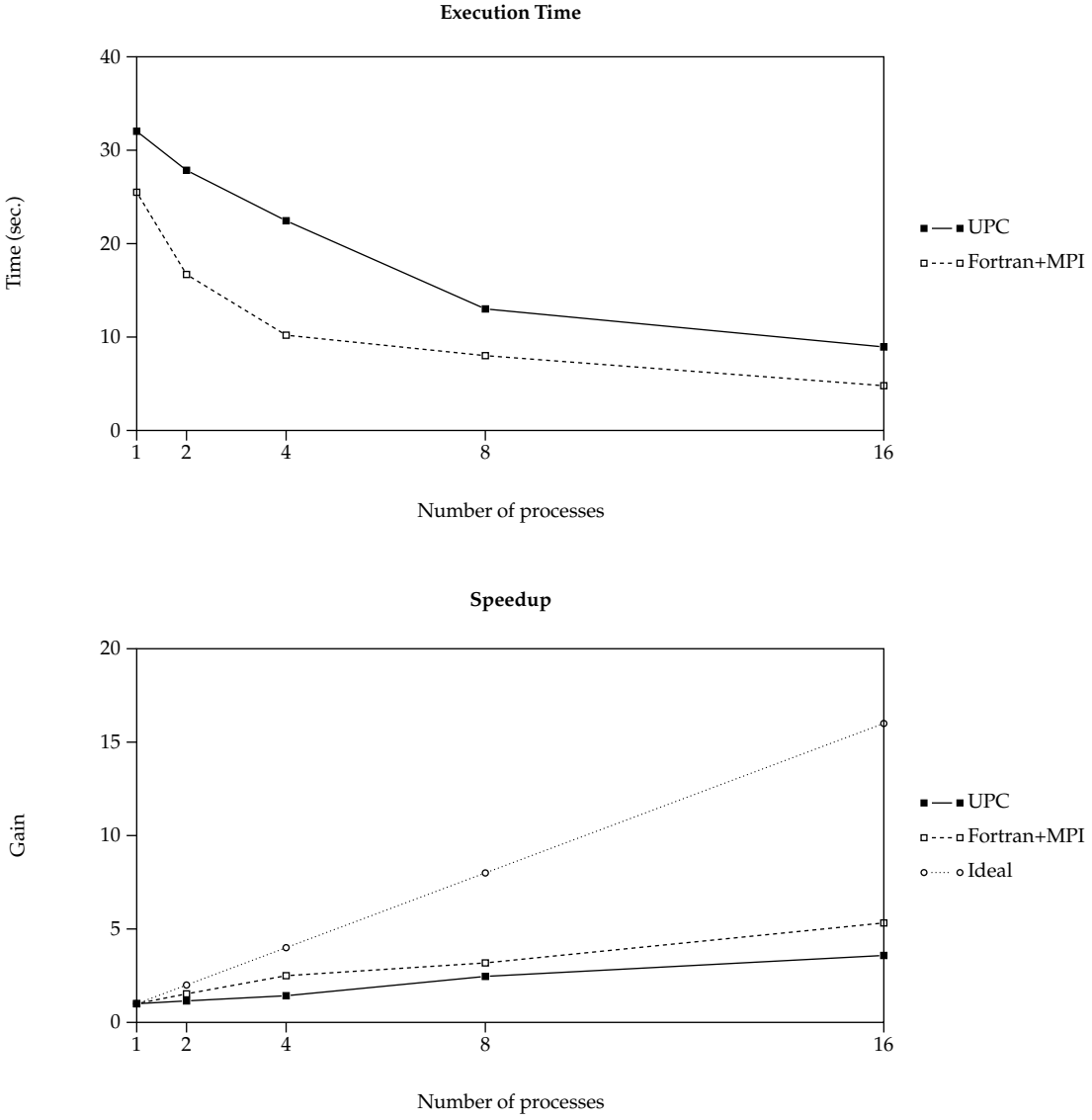
4. Future of the UPC Project

It is very likely that Unified Parallel C will soon have a leading position in parallel programming. From the beginning, the UPC has had the support of governmental agencies as well as major companies in the field of parallel computing. This partnership with companies has permitted to have mature or almost mature implementations of UPC compilers less than one year after the specifications of the language were released. One of these implementations has recently been publicly released and the other one is freely available from GW's website since Novenber 2001; three other implementations from other vendors are currently under active development.

As of the beginning of the year 2002, the UPC language is still defined by version 1.0 of the UPC Specifications [8]. However, there are quite a lot of major changes that are expected to be added this year to the specifications. First of all, the UPC Consortium has already scheduled a Workshop, that is a gathering of all the people involved in the Project that should take place in March 2002 in Washington, DC. Several important issues regarding the language are to be discussed during this workshop among which the most important are the design of a parallel input/output library as well as the design of a collective operations library.

The next version of the specifications should fill the gap that has existed between UPC and, for instance, MPI. UPC's programming paradigm seems currently to be both easy from the programmer's point of view and designed in a way that should match most parallel architectures. Considering the ongoing work on the UPC Specifications as well as on the various implementations, we can expect UPC to become more and more widely used very quickly.

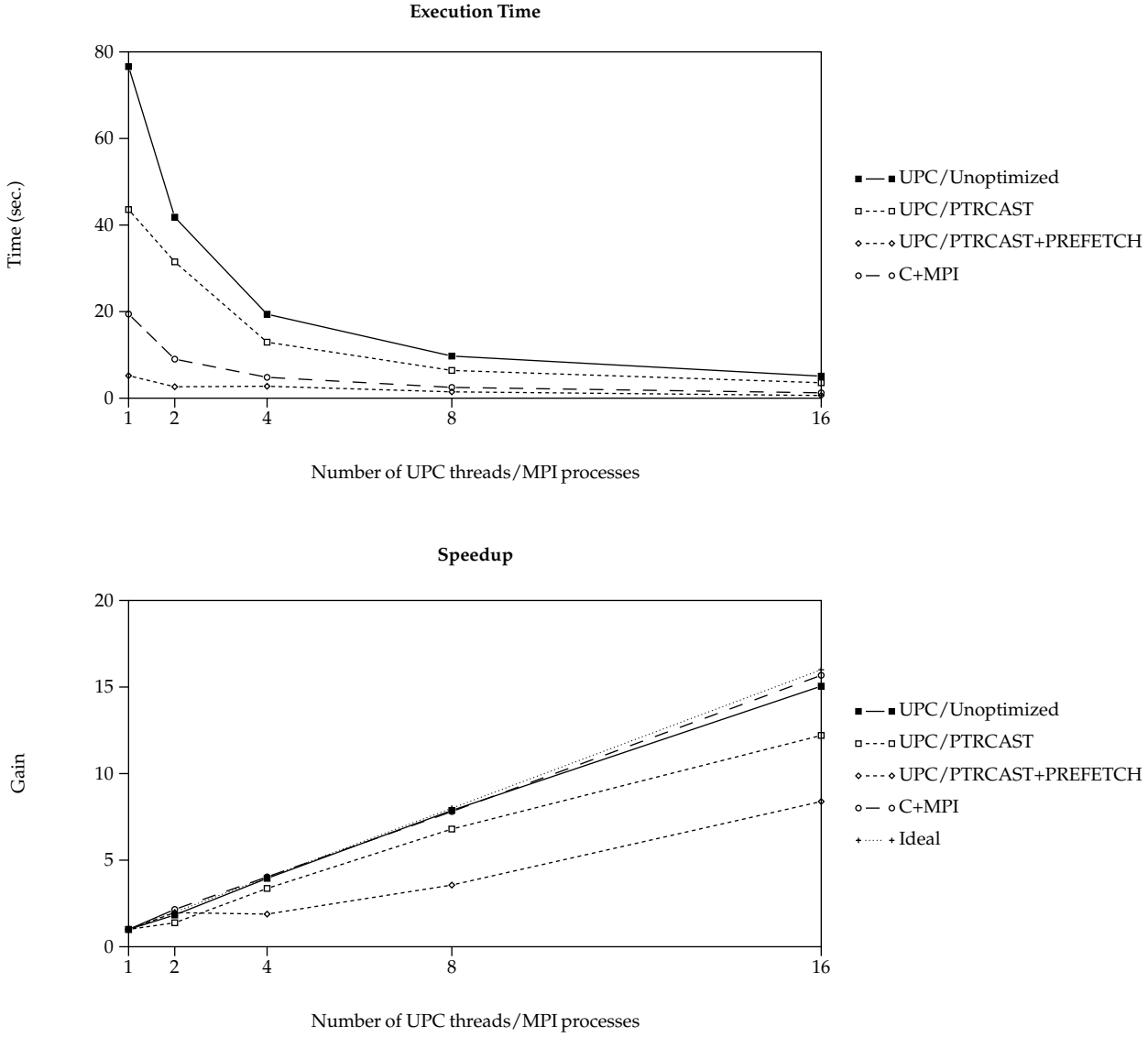
Appendix A. Performance of the FT kernel, class A, on the Compaq platform



The FT kernel of the NAS Parallel Benchmark requires a lot of communication among the threads. As a consequence, the performance curves above show a signifi cant gap between the ideal speedup and the actual speedup of this application.

Appendix B. UPC_Bench on the SGI Origin 2000

Performance of the matrix-multiplication application with square matrices of 512 per 512 integers[1].



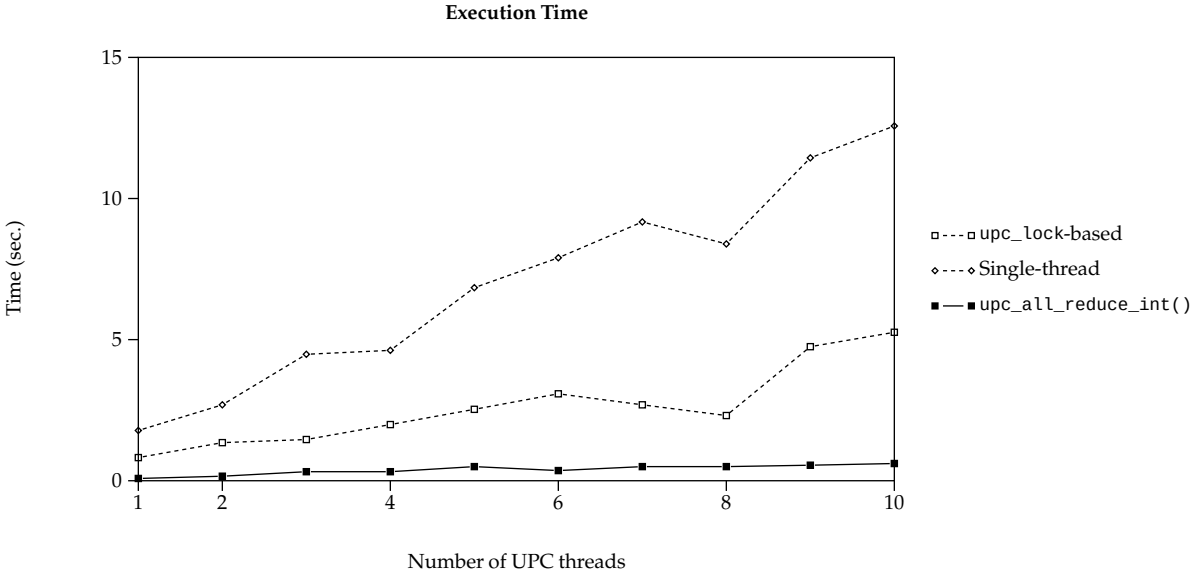
UPC/Unoptimized denotes results obtnained with standard UPC code, i.e. without any hand-optimization, whereas *PTRCAST* represents the "pointer casting of local shared pointers to private pointers" optimization and *PREFETCH* represents the prefetching of remote data. *C+MPI* shows the performance of the same parallel program written in C and using MPI.

[1]Using GNU-UPC 1.8, GCC-2.95.2 based

Appendix C. The UPC Reduction Function `upc_all_reduce`

C.1. Performance Comparison of `upc_all_reduce`

Addition of arrays of 1 million 32-bit integers on the SGI Origin 2000 (running Irix 6.5), up to 10 UPC threads, using GNU-UPC 1.8, GCC-2.95.2 based, and which uses the Mips2 32-bit ABI[1].



The two implementations that are compared to `upc_all_reduce()` use very simple methods that do not actually perform well. The one marked as *single-thread* has only one thread doing the computation, that is, one thread fetching the arrays from other threads and adding them to the final array. The one marked *Lock-based* uses `upc_lock`s (described in section 3.1.5) and is available in subappendix C.2.

[1]Application Binary Interface

C.2. Example code of a non-optimized addition of arrays among threads

```
shared[] ELEM_T*
upclock_sum(const ELEM_T* myop, size_t size,
            upc_operator op, ELEM_T (*f)(ELEM_T,ELEM_T),
            size_t thr)
{
        static int first_time=1;
        size_t i, j;
        shared[] ELEM_T* ret=NULL;
        shared[] ELEM_T* shared *pret;

        if(first_time)
        {       upc_lock_init(&sumlock);
                first_time=0;
                upc_barrier;
        }

        /* Allocates an array for the result (ret) having affinity to thread #thr.
         * The pointer to this array is broadcasted to all the threads such that
         * MYTHREAD!=thr via 'pret'.
         */
        pret =
                (shared[] ELEM_T* shared*)upc_all_alloc(1, sizeof(shared[] ELEM_T*));
        assert(pret != NULL);
        if(MYTHREAD==thr)
        {       ret = upc_local_alloc(size, sizeof(ELEM_T));
                assert(ret != NULL);
                (*pret) = ret;
                for(j=0; j<size; j++) ret[j]=0;
        }
        upc_barrier;
        /* Broadcast the pointer to the return value to each thread */
        if(MYTHREAD!=thr) ret = *pret;
        upc_barrier;

        /* Compute the sum */
        upc_lock(&sumlock);
        for(i=0; i<size; i++) ret[i]+= myop[i];
        upc_unlock(&sumlock);

        upc_barrier;
        return ret;
}
```

References

[1] Website of the ECE Department of the School of Engineering and Applied Science (SEAS) of the George Washington University.. URL *http://www.ece.seas.gwu.edu/*.

[2] MPICH Website. URL *http://www-unix.mcs.anl.gov/mpi/mpich/*.

[3] Libby Wiebel. A Modified Proposal for the UPC Collective Operations. Tech. Rep. (June 2001). URL *http://upc.gwu.edu/ (Developers Area)*.

[4] Sébastien Chauvin and Tarek El-Ghazawi. Proposal for the UPC Collective Operations. Tech. Rep. (2001). URL *http://upc.gwu.edu (Developers Area)*.

[5] Sébastien Chauvin and Tarek El-Ghazawi. *UPC Benchmarking Issues*, pages 365-372. International Conference on Parallel Processing, 2001, 2001.

[6] The Message Passing Interface Forum (MPIF). MPI: A Message Passing Interface. Tech. Rep. (June 1995). URL *http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.html*. MPI Standard 1.1

[7] William W. Carlson and Jesse M. Draper. *Distributed Data Access in AC*, pages 39-47. Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara, July 19-21 1995.

[8] William W. Carlson, Jesse M. Draper, Tarek A. El-Ghazawi. UPC Language Specifications V. 1.0. Tech. Rep. (February 2001). URL *http://upc.gwu.edu/*.

Index