Unified Parallel C Profiling Interface Proposal

Adam Leko, leko@hcs.ufl.edu
UPC Group, HCS Lab
University of Florida

July 2, 2005

Abstract

Due to the wide range of compilers and the lack of a standardized profiling interface, writers of performance tools face many challenges when incorporating support for Unified Parallel C (UPC) programs. This document presents a preliminary specification for a standard profiling interface that attempts to be flexible enough to be adapted into current UPC compiler and runtime infrastructures with little effort, while allowing performance analysis tools to gather much information about the performance of UPC programs.

1 Introduction

The Unified Parallel C (UPC) [6] language offers parallel programmers several advantages over other parallel languages that require programmers to manually handle communication between nodes. The global address space presented to UPC programmers provides them with a convenient environment similar to threaded programming on serial machines, but comes at a cost of increased complexity in UPC compilers and runtime systems. This gives parallel programmers a much-needed increase in productivity; however, since UPC compilers handle much of the low-level communication and work distribution details of UPC programs, it can be difficult or impossible for UPC programmers to determine how a given UPC will perform at runtime.

Indeed, recent research has indicated that performance tuning for current generations of UPC compilers is absolutely critical in order to achieve comparable performance to MPI code, especially on cluster architectures [5]. Although recent work has produced several techniques to improve the performance of compilers by taking advantage of specific architectures [1] or employing TLB-like lookup tables for remote pointers [4], the UPC programmer's ability to exploit locality remains the most influential factor on overall UPC program performance.

The importance of performance analysis for UPC programs has also been aggravated by the lack of performance analysis tools supporting UPC. The newness of the UPC language is partly responsible for the lack of tool support, but tool developers face a major roadblock even if they wish to add UPC support in their tools: the UPC specification currently does not define a standard profiling interface that can be used to portably gather performance information from UPC programs at runtime. The extensive and almost exclusive use of the MPI profiling interface [7] by MPI performance tools illustrates the usefulness of a common profiling interface.

To rectify this situation, the UPC group at the University of Florida has developed a preliminary profiling interface to accompany the UPC specification. In a nutshell, we are trying to help users answer the question "Why does my UPC program have bad performance?" by providing tool developers with a consistent profiling interface so that their performance tools can help users identify and fix performance bottlenecks. We have examined several existing UPC implementations and have attempted to devise an interface that will have a minimum impact on existing compilers that target both shared-memory and distributed architectures. Since UPC is somewhat similar in spirit to OpenMP, we have tried to incorporate many ideas used for the OpenMP profiling interface [9] in our proposal.

Note: In order to keep the language of this document concise, we have adopted the following terminology. "UPC" refers to the Unified Parallel C language as defined by the current language specification. "Users" refers to individuals using a parallel language such as UPC. "Developers" refers to individuals who write parallel software infrastructure such as UPC compilers. "Tools" refers to performance analysis tools such as Vampir, TAU, or KOJAK, and "tool developers" refers to individuals who write performance analysis tools.

2 Current challenges for UPC tool developers

MPI and other library-based parallel language extensions generally use techniques like weak bindings, which make developing "wrapper" libraries very straightforward. For example, Figure 1 illustrates a typical wrapper that can be use to profile

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
  int val;
  Record_message(comm, dest, count, buf);
  Timer_start("MPI_Send");
  val = PMPI_Send(buf, count, datatype, dest, tag, comm);
  Timer_end("MPI_Send");
  return val;
}
```

Figure 1: Example wrapper function

calls to `MPI_Send` for MPI programs. MPI tool developers can rely on the existence of a `PMPI_Send` binding in every MPI implementation which has the same semantics as the original `MPI_Send` function. Tool developers may place any code they wish in their wrapper libraries, which allows them to support tracing, profiling, and several other analysis strategies. In addition, users of MPI tools need only to relink against a profiled MPI library in order to collect information about the behavior of MPI functions in their code.

UPC tool developers are not as lucky. Since UPC is not merely comprised of library functions, the wrapper approach mentioned above does not present a tractable (or even valid) strategy for recording the behavior of UPC programs. Even for UPC implementations that encapsulate much behavior in library functions (such as Berkeley UPC [8] with GASNET [2]), no weak binding support is available to write wrappers for these libraries. In addition, different UPC compilers use vastly different techniques to translate UPC code into executables, ranging from source-to-source transformations to more direct compilation techniques. This leaves the UPC tool developer with only a few methods for collecting information about a UPC program: binary instrumentation and source instrumentation.

Binary instrumentation seems like a decent option at first glance. Libraries like DynInst [3] make it relatively easy to insert arbitrary instructions at points in a user's code, and since the user's source code does not have to be changed this reduces the overhead imposed upon users. However, in order to insert instrumentation code, binary instrumentation requires tool developers to explicitly state *exactly* where they wish to insert their code in an executable. Since UPC compilers can use any one of a number of techniques to translate UPC code into an executable, binary instrumentation for UPC programs will have to be added on a compiler-by-compiler basis after an extensive analysis of how that particular compiler translates the relative UPC constructs into executable code. This presents a severe maintenance problem for tool developers: what happens when developers change their implementation internals between versions?

In short, tool developers adopting a binary instrumentation strategy will have to spend a large portion of their time keeping up with the internal workings of each UPC compiler they wish to support. In addition, binary instrumentation makes it very difficult to relate performance information back to actual lines in source code. Another blow to binary instrumentation is the possibility that choosing instrumentation points in UPC binaries may be impossible without proprietary information from developers of commercial UPC compilers. Finally, while DynInst provides great functionality for tool developers wishing to use binary instrumentation, there are no plans to port it to Cray architectures such as the X1 that are not based on Linux. The X1 and X1E provide excellent vehicles for running UPC code [1], so excluding tool support for these architectures would be disappointing to say the least.

Source instrumentation affords the most flexibility to the tool developer. With source instrumentation, a tool developer can record any data they wish, provided that the information can be extrapolated from a user's code. However, with the increasing complexity of UPC runtime systems and the use of techniques such as remote reference caching, using source instrumentation alone limits a tool developer's ability to actually find out when certain events *actually* occur instead of when the tool developer *thinks* they might occur. UPC's relaxed memory model further blurs the distinction between the sequence of actions specified in the UPC source code and what actually happens at runtime.

Source instrumentation also places a higher technical burden on the tool writer. Since UPC allows users to treat remote pointers in the exact manner as built-in C pointers, source code instrumentation systems must be able to perform a *full* parse of a user's UPC source code so that remote variable accesses can be differentiated from local variable accesses. Several complications arise when doing a full parse, including variable name shadowing (which implies that the parser must fully implement scoping rules), complicated user macros, and semantic analysis of complicated expressions involving both local and shared variables. To give a concrete example of the possible difficulties in parsing a complicated expression, consider the code in Figure 2. Depending on the algorithms used to translate the UPC code into machine code, several different sequences of remote read and write operations are possible. In addition, compilers which can do complicated strength

```
#include "upc.h"

shared int c = 44;
shared int f = 2;

int main() {
  f = ((c + 3) - (c - 3) == c) ? 4 : (c = 0);
  return 0;
}
```

Figure 2: UPC code with a complicated expression

reductions may be able to reduce the first line of `main` to `f = 0; c = 0;`, which drastically reduces the number of remote memory references.

Clearly, neither source nor binary instrumentation represent optimal paths for tool developers that wish to add UPC support to their tools. Having a standard profiling interface into the UPC compiler and runtime systems would make tool developer's jobs much easier and would allow them to easily add support for any compiler that implements the interface. We describe our ideas for such an interface in the next section.

3 Proposed profiling interface

In order for a profiling interface to be effective, it must meet several criteria and strike a balance between functionality for tool developers and ease of implementation for compiler developers. We have designed a basic profiling interface that tries to follow these design guidelines:

- *Flexibility* – In order for the interface to be most useful, it must be flexible to support several different performance analysis methods. For example, some tools rely on capturing full traces of a program's behavior, while others calculate statistical information at runtime and display it immediately after a program finishes executing. Our profiling interface should support these two main modes of operation, and should not overly restrict the tool developer's analysis options.

- *Ease of implementation* – It is imperative to have the full support of UPC compiler developers for our profiling interface. There are many existing

UPC compilers that translate UPC code into executables using a wide variety of methods, and we do not wish to favor one compilation approach over the others. We do not wish to alienate compiler writers by making our profiling interface difficult to implement for a small group of unlucky compilers. In addition, many UPC compilers are proprietary, which limits the amount of help we can give in implementing the profiling interface. Therefore, our interface should be as implementation-neutral as possible while meeting the rest of our design goals.

- *Low overhead* – Our profiling interface should not drastically affect overall runtime of profiled UPC programs. Performance data collected for programs that do not exhibit the same behavior as their unprofiled counterparts is not very useful to users. Due to the fine-grained nature of most UPC programs, we do expect some perturbation of overall execution time for profiled programs. However, at every opportunity we wish to engineer solutions that minimize the effect instrumentation has on a user's program.

- *Usefulness* – Simply meeting the above three goals without providing useful information to users will still result in a tool that does not help a user troubleshoot their performance problems. Performance tools need to be provided with enough information to analyze so that the tools can present the user with potential problem areas in their application codes. Specifically, we feel it is absolutely necessary to incorporate source code correlation for data reported to the user down to the source line level.

In addition to the above design guidelines, we shall favor simpler solutions over more complex ones. We do not wish to over-engineer a solution that could be obtained by simpler methods. In other words, our profiling interface should be as simple as possible to meet the above guidelines, but no simpler.

We envision our profiling interface working in the following manner. First, the user compiles their code using special compilation flags that insert standard function calls into the user's code as defined by our profiling interface, along with the standard translation that occurs as part of a normal UPC compilation process. The user links their code against a library provided by a tool developer that implements part of the profiling interface, then runs their program. During runtime, when certain events happen, appropriate function calls are made to the tool developer's library code, which produces data files or output to the screen (depending on how the tool developer writes their profiling library).

The first half of our profiling interface is a set of functions implemented by tool developers which are akin to callback functions. The UPC runtime system invokes these functions when certain events happen during runtime, and tool developers decide what information to record and what actions to take. These functions are defined and discussed in Section 3.1. The second half of our profiling interface is a set of functions tool developers can use to gather information from the UPC runtime and compiler systems. These functions are defined and discussed in Section 3.2.

3.1 Tool-provided functions

Using a callback model for notifying tools about when certain activities occur affords the tool developer much flexibility and shifts the responsibility of tracking performance data from the compiler writers to the tool developers (where it belongs). While inserting function calls does increase overall execution overhead slightly, it is a much simpler alternative than trying to have compilers inline profiling code inside a user's program, which would create a very challenging static analysis problem.

Our interface proposes two main types of callback functions: simple wrappers and start/end function pairs. Simple wrappers are appropriate for the utility UPC functions (such as the memory and lock allocation/deallocation routines) since this part of the language already exists in the form of functions. Start/end function pairs are more appropriate for constructs and other parts of the language that do not follow the function format, such as the synchronization statements (like `upc_notify` and `upc_wait`) and language constructs (like `upc_forall`).

Trying to provide a wrapper interface for non-function language constructs would be extremely difficult, since a unified interface would have to be exported from compilers so that the original functionality of these activities could be called from within the wrapper. For example, trying to wrap the `upc_forall` would be problematic; what is an appropriate way for the wrapper library to let the UPC runtime know it should execute a particular `upc_forall` loop? Therefore, we feel it is appropriate to notify tools when the particular statements start and finish executing, and also provide these functions with extra information that lets them relate timing information back to the source-code level. A very similar approach is taken by POMP, OpenMP's profiling interface, which is exploited by OPARI [9].

We propose that the callback and wrapper functions be allowed to use all available UPC language constructs and functions in order to ease the tool developer's job. This could be easily accomplished by compiling the wrapper library itself using a UPC compiler. This should also enable compiler writers to implement the functions defined in the UPC specification any way they wish, including inlining the code for functions such as `upc_memget`, without sacrificing support for the profiling interface.

The rest of this subsection describes the wrapper function and start/finish function callback pairs for each major part of the UPC language: synchronization statements, language constructs, utility functions, and implicit communication. Note that all of the `pupc*` functions take an extra argument as compared with their `upc*` equivalents, which is a pointer to a type `pupc_location`. This type is a struct that contains source code information, and is defined as below:

```
int pupc_num_locations;
struct pupc_location {
  const char* source_file;
  unsigned int start_line;
  unsigned int end_line;
};
pupc_location* pupc_locations;
```

The information contained in this struct should point to the location of the callsite or construct location in the original (untranslated) version of the user's code. The `pupc_locations` is a pointer to an array of size `pupc_num_locations` that lets a tool easily iterate over all existing instrumentation points in a user's program[1]. By allocating this source code information statically inside an instrumented executable, source correlation can be achieved at relatively low CPU and memory cost. Also note that by passing pointers to each function, a tool can easily differentiate between two or more of the same constructs on a single source line.

3.1.1 Synchronization callback functions

Function prototypes:

```
void  pupc_notify(int, pupc_location*);
void  pupc_wait_start(int, pupc_location*);
void  pupc_wait_end(int, pupc_location*);
void  pupc_barrier_start(int, pupc_location*);
```

[1]This may be difficult for compiler writers to perform that do not have fine-grained control over the linking phase of a program

```
void  pupc_barrier_end(int, pupc_location*);
void  pupc_fence_start(pupc_location*);
void  pupc_fence_end(pupc_location*);
```

These functions are inserted into the user's code and get called at runtime before and after execution of the notify, wait, barrier, and fence synchronization statements. Since `upc_notify` is a nonblocking operation, only one callback function is needed. The first argument to the notify, wait, and barrier should be the value of evaluating the optional integer expression; if the user has not specified one, a value of zero should be passed to these functions.

3.1.2 Language construct callback functions

Function prototypes:

```
void  pupc_forall_start(pupc_location*);
void  pupc_forall_end(pupc_location*);
```

These functions are inserted before and after `upc_forall` constructs and get called at runtime before and after the `upc_forall` construct executes.

3.1.3 Utility function wrappers

Function prototypes:

```
void         pupc_lock(upc_lock_t*, pupc_location*);
void         pupc_unlock(upc_lock_t*, pupc_location*);
int          pupc_lock_attempt(upc_lock_t*, pupc_location*);
void         pupc_lock_init(upc_lock_t*, pupc_location*);
upc_lock_t*  pupc_all_lock_alloc(pupc_location*);
upc_lock_t*  pupc_global_lock_alloc(pupc_location*);
void         pupc_lock_free(upc_lock_t*, pupc_location*);
void         pupc_memcpy(shared void*, const shared void*,
                         size_t, pupc_location*);
void         pupc_memget(void*, const shared void*,
                         size_t, pupc_location*);
void         pupc_memput(shared void*, const void*,
                         size_t, pupc_location*);
void         pupc_memset(shared void*, int,
                         size_t, pupc_location*);
void         pupc_init(int*, char***);
void         pupc_global_exit(int status, pupc_location*);
```

These function wrappers are exactly the same as their counterparts defined in the UPC specification, with the slight name change and extra argument that is a pointer to the source code location for each invocation. We have also included a generic `pupc_init` that gets called at the beginning of every execution after the UPC runtime library has been initialized so that tool developers may easily include any initialization code they require. We recommend that the UPC runtime call `pupc_global_exit` at the end of program execution with a NULL value for the pupc_location pointer, even if the user makes no calls to `upc_global_exit`. This will enable tool developers to ensure that any cleanup code they require will always get executed just before a UPC program finishes execution.

3.1.4 Implicit communication callback functions

Function prototypes:

```
void  pupc_remote_get_start(shared void*, void*, int,
                            size_t, pupc_location*);
void  pupc_remote_get_end(shared void*, void*, int,
                          size_t, pupc_location*);
void  pupc_remote_put_start(shared void*, void*, int,
                            size_t, pupc_location*);
void  pupc_remote_put_end(shared void*, void*, int,
                          size_t, pupc_location*);
```

These functions are inserted into a user's code before and after singular remote memory transfers take place. The first argument to these functions is the shared pointer to the remote variable being read from or written to. The second argument is a pointer to a local memory address that the source location is being read into or written from. The third argument is an integer value specifying if the transfer taking place is being performed under the relaxed memory model or not. A zero value indicates that the strict memory model has dictated this memory transfer, while a nonzero value indicates that the variable is using the relaxed memory model under the duration of this transfer. Finally, the last two arguments give the size of the transfer and a pointer to the source code location that caused this remote transfer.

3.2 Compiler-provided functions

While the above provides tools with a large amount of information, it would be beneficial for the UPC compiler and runtime system to have a few utility routines that may be used to enhance the information the tool can gather. First, it would be very advantageous to have a global wallclock timer, similar to MPI's `MPI_Wtime` function, so that assigning timestamps to events is more convenient for the tool developer. Second, it would extremely useful to have a generic routine that tools could use to answer queries about arbitrary shared variables, such as determining a variable's blocking factor, size, type, and the source code location of its definition.

Function prototypes and related structs for these two functions are shown below[2]:

```
unsigned long long pupc_wtime();
enum pupc_variable_type {
  ptr, int, char, long, float, double
};
struct pupc_variable_info {
  const char* name;
  enum pupc_variable_type type;
  size_t total_size;
  unsigned int array_size;
  unsigned int blocking_factor;
  pupc_location* source_location;
};
void pupc_query_variable(shared void*,
                         pupc_variable_info*);
```

4 Examples

To provide a concrete example of how our profiling interface would work on an actual program, we present a example of how a compiler might translate a simple UPC program by adding appropriate calls to the profiling functions.

Consider the simple UPC program listed below, with line numbers included for reference:

[2]I'm not sure how to handle querying the variable type here. The enum was all that I could come up with. Also, I don't know how difficult it would be to do something like the pupc_query_variable, but it would definitely be incredibly useful for performance tools.

```
#include "upc.h"

shared int a[5*THREADS];

shared int b = 0;

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

  upc_fence;
  sleep(1);
  upc_forall(i = 0; i < 5*THREADS; &a[i]) {
    a[i] = 44;
    if (MYTHREAD == 1) {
      b = i;
    }
  }
  return 0;
}
```

Based on the interface defined in the previous section, a source-to-source compiler might translate the code to the following UPC code (assuming it realizes that the first assignment of the `upc_forall` is always a local assignment):

```
#include "upc.h"
#include "pupc.h"

shared int a[5*THREADS];

shared int b = 0;

struct pupc_location loc1, loc2, loc3;
int pupc_num_locations = 3;
struct pupc_location* pupc_locations;

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

  pupc_locations = (pupc_location*)malloc(
    sizeof(pupc_location) * pupc_num_locations
  );
  pupc_locations[0] = loc1;
  pupc_locations[1] = loc2;
  pupc_locations[2] = loc3;

  loc1.source_file = __FILE__;
  loc1.start_line = 10;
  loc1.end_line = 10;

  loc2.source_file == __FILE__;
  loc2.start_line = 12;
  loc2.end_line = 17;

  loc3.source_file == __FILE__;
  loc3.start_line = 15;
  loc3.end_line = 15;

  pupc_init(&argc, &argv);

  pupc_fence_start(0, &loc1);
  upc_fence;
  pupc_fence_end(0, &loc1);

  sleep(1);

  pupc_forall_start(&loc2);
  upc_forall(i = 0; i < 5*THREADS; i++; &a[i]) {
    a[i] = 44;
    if (MYTHREAD == 1) {
      pupc_remote_put_start(&b, &i, 0,
                            sizeof(int), &loc3);
      b = i;
      pupc_remote_put_end(&b, &i, 0,
                          sizeof(int), &loc3);
    }
  }
  pupc_forall_end(&loc2);
```

```
        pupc_global_exit(0, NULL);
}
```

5 Conclusions

We have presented a profiling interface for UPC codes that will enable tool developers to easily add support for UPC programs in their tools. The interface is flexible enough to support both profiling and tracing UPC programs, and should impose only a slight overhead on compiler developers.

While we have strived to make our profiling interface as lightweight as possible, there will still be some unavoidable overhead that will occur on fine-grained UPC programs that use a large number of remote accesses. This will generally only be an issue for programs that only run well on shared-memory machines.

In order for a performance tool to be useful it must be able to get an accurate reading of a program's actual runtime behavior. While hardware counters may be used on some platforms to gauge how many remote accesses are occurring for a UPC program, hardware counters can only provide a very rough approximation of a UPC program's behavior. In addition, it is nearly impossible to relate the hardware counter information down to the source-code level for UPC programs without having a profiling interface like the one proposed in this document.

Since there will undoubtedly be some unavoidable overhead associated with the use of a fully-profiled UPC program, we suggest that compilers provide two flags that may be used at compile time to decide which parts of a user's program to instrument. One flag, say `--profile`, will be used to specify that the compiler add profile hooks for *all* of the interface as defined in this document, while another flag, say `--profile-partial`, profiles everything except for implicit communication. This should enable performance tools to record accurate performance data even for fine-grained UPC programs on shared-memory architectures, even though the amount of data recorded will be incomplete. We feel that recording performance data for the `upc_forall` construct and recording timing information spent in barriers will still provide users with adequate information to tune their UPC code.

Finally, we welcome (and encourage!) any constructive criticism of this proposal, especially from compiler writers. We have made some assumptions in this document, and if they are incorrect please let us know! We would like to work as closely as possible with UPC compiler developers to ensure that the UPC profiling interface becomes a reality.

References

[1] Christian Bell, Wei-Yu Chen, Dan Bonachea, and Katherine Yelick. Evaluating support for global address space languages on the Cray X1. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, New York, NY, USA, 2004. ACM Press.

[2] Dan Bonachea. GASNet specification, v1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[3] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] François Cantonnet, Tarek A. El-Ghazawi, Pascal Lorenz, and Jaafar Gaber. Fast address translation techniques for distributed shared memory compilers. In *IPDPS*. IEEE Computer Society, 2005.

[5] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, Fran&#231;ois Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarr&#237;a-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM Press.

[6] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC language specification (v 1.2), June 2005.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[8] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the Berkeley UPC compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM Press.

[9] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *J. Supercomput.*, 23(1):105–128, 2002.