SAND REPORT

SAND2006-0422
Unlimited Release
Printed Jan. 2006

Some Language Issues in High Performance Computing: Translation from Fine-grained Parallelism to Coarse-grained Parallelism

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-0422
Unlimited Release
Printed Jan. 2006

Some Language Issues in High Performance Computing: Translation from Fine-grained Parallelism to Coarse-grained Parallelism

Sue Goudy*, Shan Shan Huang, Zhaofang Wen†

Abstract

A parallel programming model, BEC, was proposed in [1], to enable Global Address Space (GAS) capabilities for programming in SPMD style. It is a portable light-weight approach for incremental acceptance of the GAS model, along an evolution path that leverages existing infrastructures. It assists migration of legacy applications thereby encouraging their expert programmers to adopt the new model. BEC also provides for some unaddressed needs, such as efficient support for high-volume fine-grained random communications. BEC can be used as an enhancement to existing environments such as MPI.

This report presents a scheme for a compiler to translate high level GAS languages PRAM C ([2]) and UPC ([9]) into BEC. Since PRAM C implements the theoretical Parallel Random Access Machine (PRAM) model [5] and BEC implements the Bulk Synchronous Parallel (BSP) model [10], such a translation by a compiler is theoretically significant, because it is the first time that a program in PRAM semantics (fined-grained parallelism) is translated into program in BSP style (coarse-grained parallelism). This provides a bridge from the PRAM model (considered impractical) to the BSP model (considered practical). Because BEC leverages infrastructures on existing platforms (such as

*email: spgoudy@sandia.gov
†email: zwen@sandia.gov

MPI or SHMEM [7, 6]), this translation scheme enables a new and alternative approach for higher level GAS language implementations which can avoid heavy investment in re-inventing much of the same communication capabilities.

Acknowledgment

The authors gratefully acknowledge the support and encouragement from our managers, Sudip Dosanjh, Neil Pundit, and Jim Ang. Funding for students to work on the PRAM C translations project was provided by Sandia's Computer Science Research Institute, for which we are indebted to Director David Womble. Numerous discussions with Jonathan Brown contributed strongly to the technical content of this report. Thanks to Ron Brightwell, whose thorough review comments greatly improved this presentation.

We wish to acknowledge distinguished leaders for their contributions in the field and for inspiring us. These notably include Bill Carlson, Kathy Yelick, and Bob Numrich. In particular we gratefully acknowledge Bill Carlson's guidance and his assimilation of Zhaofang Wen and Mike Heroux into the UPC Consortium. Thanks also to the UPC Consortium and, in particular, Lauren Smith, Tarek El-Ghazawi, Phil Merkey, Steve Seidel, and Dan Bonachea.

Special thanks to Lewis Goudy for proofreading the document, which greatly enhanced its presentation.

Contents

Some Language Issues in High Performance Computing: Translation from Fine-grained Parallelism to Coarse-grained Parallelism

1 Introduction

On distributed memory hardware, Global Address Space (GAS) provides ease of programming by allowing one processor direct access to another processor's memory. GAS promises improved productivity over existing programming environment (such as MPI); and therefore it is attractive for high productivity computing systems (HPCS).

A parallel programming model, BEC ([1]), was introduced recently to enable GAS capabilities for parallel programming in SPMD style. It is a portable light-weight approach for incremental acceptance of the GAS model, along an evolution path that leverages existing infrastructures and maintains backward compatibility with existing programming methods and environments. It assists migration of legacy applications thereby encouraging their expert programmers to adopt the new model. In addition, BEC provides for some of the unaddressed needs, such as efficient support for high-volume fine-grained and random communications, which are common in parallel graph algorithms, sparse-matrix operations, and large scale simulations.

This report presents a scheme for a compiler to translate high level GAS languages PRAM C ([2]) and UPC ([9]) into BEC. Such a scheme is important for the following reasons.

- PRAM C adopts the semantics of the of the theoretical PRAM model (for Parallel Random Access Machine [5]); and BEC implements the BSP model [10]. This kind of translation by a compiler, from PRAM C to BEC, is theoretically significant, because it is the first time that a program in PRAM semantics (fined-grained parallelism and/or communication) is translated into a program in BSP style (coarse-grained parallelism and communication).

  This provides a real bridge from the theoretical PRAM model (considered impractical) to the BSP model (considered practical).

- BEC provides the appropriate level of abstraction not only for GAS style programming, but also for use as an intermediate GAS language to which other higher level GAS languages can be compiled or translated.

- Fine-grained communication (through virtual shared memory in PRAM C and in UPC) can be translated into coarse-grained communication (in BEC).

- Translating PRAM C into BEC provides an alternative approach to implementation of PRAM C, versus the thread-based approach in [2]. This approach avoids the overheads and potential limitations associated with management of a possibly huge number of threads. This also enables many compiler optimization techniques to be applied to improve the PRAM C program performance.

- Translating UPC into BEC also provides an alternative approach to implementation of UPC. As a result, the fine-grained communication in a UPC program can be converted to coarse-grained communication in actual implementation.

- More generally, this translation scheme enables a new and alternative approach for higher level GAS language implementations which can avoid heavy investment in re-inventing much of the same communication capabilities.

2 Features of PRAM C

PRAM C is a simple extension to ANSI C that adds constructs and data types to support the PRAM model. The control constructs support both Single Instruction Multiple Data (SIMD) and task parallel programming styles. The shared data type adds the capability to access remote data implicitly by simply referencing that data. Delivery of the remote data is handled by the PRAM C runtime support structure. Details of PRAM C can be found in [2], here we present a brief overview of the language.

2.1 Parallel Control Constructs

- **PRAM_do**: This construct takes as argument an integer K. This construct indicates that function $f(...)$ will be executed by K virtual PRAM processors. K is a way to express the degree of fine-grained parallelism in the algorithm that the programmer wants the computer system to exploit.

```
PRAM_do(K): f(...);
```

The theoretical PRAM model requires a SIMD style of execution of its instructions; that is, execution of every instruction is synchronized. Inside a **PRAM_do** construct, execution of the statements follows a Single Program Multiple Data (SPMD) model; and synchronization is required only at the end of **PRAM_do** construct and also at (implicit) barriers as described next. This relaxation should help performance.

No explicit barrier should be necessary inside **PRAM_do**. However, there are implicit barriers within such a construct. These implicit barriers are honored by the group of virtual PRAM processors within PRAM_do construct:

- The programmer can use the ANSI C block statement syntax (statements enclosed within curly braces {...}) to indicate that the execution of the block of code needs to be synchronized, implying a group barrier at the end of the C block.
- Any simple or compound statement implies a group barrier at the end whenever a shared array access appears in the statement. This should be sufficient to support the SIMD style of synchronization, because program correctness can only be affected by out-of-order shared memory accesses.

Note that these implicit group barriers can be handled by the thread scheduling scheme, and thus do not require explicit synchronization [2]. (If PRAM C is implemented based on translation into BEC (as discussed in this paper), the implicit barriers are implemented indirectly through the calls to BEC_exchange().)

- **PRAM_fork_join**: This indicates that C functions $f1(...)$, $f2(...)$, ... will be called in parallel.

```
PRAM_fork
   C_function:  f1(...);
   C_function:  f2(...);
     ...
PRAM_join;
```

Different branches of execution of a PRAM_fork proceed independently; that is, there is no synchronization between any two separate branches. This will be useful in writing parallel recursive functions. This construct also provides a way to express Multiple Instruction Multiple Data (MIMD) style coarse-grain

parallelism; and group synchronization will be provided by runtime services when needed.

The new constructs (**PRAM_fork** and **PRAM_do**) can be nested. They can also be nested with other compound statements of ANSI C.

2.2 Data Types and Their Physical Layout

We augment the ANSI C data types with the concept of **shared**. A shared variable is accessible by all the virtual PRAM processors as well as threads executing other parts of the program. This concept is introduced to support the shared memory processing needed by the PRAM. For example, `shared double A[N];` declares a shared array that is laid out across all the physical processors with equal partitions. In this example, assuming P physical processors, elements $A[0]$ ... $A[\frac{N}{P}-1]$ will be located in the the first processor; elements $A[\frac{N}{P}]$ ... $A[\frac{2N}{P}-1]$ will be located in the second processor; and so on. Based on such a formulation, it is easy to determine in which physical processor any element of a shared array is located. This is important in the process of bundling fine-grained shared memory (array) accesses. Other layouts could be used to handle shared items, so long as it is possible to determine the location of memory from information available at runtime.

Variables that are not shared are local to the thread and thus local to the processor executing the thread. Therefore, requests to local variables do not require remote communication and can all be resolved within host processors.

Variables declared in a **PRAM_do** are also local variables. Conceptually, it helps to think that there are duplicated sets of such variables with each set belonging to the **private** memory space of one of the virtual PRAM processors. A virtual PRAM processor's accesses to its private memory certainly do not require remote communications between physical processors. This is in contrast to a virtual PRAM processor's shared memory accesses; some of these may be resolvable within the host physical processor, while others will have to be served via remote requests to other physical processors.

Note: As discussed in [2], a group of K virtual PRAM processors is introduced at the `PRAM_do(K):f(...)` to execute function $f(...)$ in parallel. Each of the PRAM processors has an identification, `MY_ID`, which is the relative rank of the virtual processor within the group; so `MY_ID` is a value in the range of $[0, K-1]$.

3 UPC Language Features

Detailed specification of the UPC language can be found in [8]. Here we list only those features that are specific to PGAS programming, i.e., the UPC `shared` variables and the `upc_forall` loop. Only these PGAS related features are of interest here for our translation scheme.

UPC shared variables are declared with key word `shared`. For example,

```
shared int A[N];
```

declares an array of N items, which is physically distributed over P physical processors. Although it is legal for a physical processor to access any shared array elements, accesses to physically remote data is more expensive than physically local data due to communication overhead. The physical layout of an array on the physical processors is based on fixed block patterns in which the block size can be specified in the program. For example, assume there are two physical processors ($P = 2$).

```
shared [2] int B[N];
```

declares an array of N items with following cyclically distributed data layout.

```
processor 0: B[0], B[1]
processor 1: B[2], B[3]
processor 0: B[4], B[5]
processor 1: B[6], B[7]
...
```

UPC provides a parallel loop to express parallelism, typically used together with UPC shared arrays. For example,

```
shared int A[N], B[N], C[N];
...
upc_forall(i = 0; i < N; i++; &A[i]) {
    C[i] = A[i] + B[i];
}
```

In this example, N loop iterations will be executed in parallel. The fourth loop parameter, $\&A[i]$, called affinity expression, is used to hint that the compiler assign the i-th iteration to the physical processor which owns array element $A[i]$.

In general, the `upc_forall` loop has the following form.

```
upc_forall(E1; E2; E3; affinity)
   body;
```

Note:

- There is no communication or synchronization between iterations. In other words, there is no dependence between different iterations so that they can be executed in parallel asynchronously until explicit barrier statements are encountered.
- There is no nesting of `upc_forall` loops; and in case of nesting, the inner parallel loop becomes a sequential loop.
- There is no parallel reduction. (Note: this issue will be addressed by the newly-released UPC Collective Operation Specification [11].)

4 Overview of BEC

BEC is a formalization of the Bundle-Exchange-Compute programming style. BEC presents its users with convenient language extensions (to ANSI C) and library calls. Details of the language and runtime support can be found in [1].

4.1 Structure of BEC Programs

Users can expect the convenience of language extensions to ANSI C, as well as library calls. Program execution is supported by the compiler and the BEC runtime. BEC follows the SPMD model, with an instance of the BEC runtime object executing on each physical processor. We sketch the structure to which BEC programs should roughly adhere, as follows:

- Issue requests for reads from shared variables.

- Issue exchange call to serve the read requests, as well as any outstanding write requests.
- Local computation resumes, with access to requested values for remote data.
- Optional barrier calls to synchronize among processors.

4.2 Language Extensions

BEC provides a virtual shared memory interface with explicit software control for data communication. The keyword `shared` is used to declare variables with storage that can span the physical memory, even in the case where this memory is distributed. Two allocation functions are provided for the management of shared data objects. Explicit data request and exchange functions, together with processor and data locality identifiers, allow the user to control data access.

In this section, we present a summary of the BEC language extensions to ANSI C.

- `shared`: keyword used to modify a C variable declaration to allow for a shared region
- `BEC_joint_allocate`: a function called by all processors to create exactly one shared region
- `BEC_local_allocate`: called by a single processor to create a shared region
- `BEC_request`: called to make shared data available for the next phase of computation
- `BEC_exchange`: causes read requests to be served and write requests to be recorded
- `BEC_barrier`: optional synchronization tool

5 Translating PRAM C

Consider the following construct ([2]),

```
PRAM_do(K): f(...);
```

This construct takes as argument an integer K. It indicates that function $f(...)$ will be executed by K number of virtual PRAM processors. The theoretical PRAM model requires a SIMD style of execution of its instructions; that is, execution of every instruction is synchronized. Here, inside a `PRAM_do` construct, execution of the statements follows a SPMD model. Synchronization is required only at the end of `PRAM_do` construct and also at (implicit) barrier as described in [2].

One approach to implementing the `PRAM_do` construct is to create K number of (light-weight) threads, each representing a virtual PRAM processor. For implementation on a real system, these threads can be evenly assigned to the P physical processors, $\frac{K}{P}$ threads each. The threads run on the physical processors synchronously according to the PRAM C semantics. Threads accessing a shared variable will be temporarily suspended until other threads also reach the same point, therefore, all the shared variable access requests can be bundled up to be served together. Although flexible for dynamic loading, this approach introduces much thread-related overhead; there is also the potential issue of having too many threads, which may overwhelm the runtime system.

An alternative approach to translate `PRAM_do(K):f(...)` into BEC; and therefore $f(...)$ will be translated into another function $f'(...)$ (written in BEC), which will be executed by each of the P physical processors (SPMD style).

In the rest of this section, we shall discuss the translation scheme.

5.1 General Idea of Translation

For each parallel step in $f(...)$, the role of the translation is to generate code for each physical processor in order to simulate the behavior of $\frac{K}{P}$ virtual PRAM processors. Therefore, the translation uses the following general idea.

- Every simple statement will be placed in a loop of $\frac{K}{P}$ iterations.
- Every branch of the control flow will be translated for simulation.
- Operators don't have to change; it is their operands that must be translated.
  - Local variables must be replicated as many times as the number of virtual processors to be simulated on each physical processor. This is because the

virtual processors are actually working on different variables in terms of physical space. So a scalar becomes a vector, with each vector element representing the local variable of the virtual processor simulated. In general, one extra dimension of size $\frac{K}{P}$ is added to the translated local variable. (For example, a pointer becomes an array of pointers; and a C structure becomes an array of structures.)

- Shared variables (including pointers to shared) remain unchanged because the virtual processors are really working on the same variable.
- `MY_ID` is a special tag in PRAM C. It always returns the virtual PRAM processor's relative ranking in the group of virtual processors employed at `PRAM_do(K):f(...)`. Here, `MY_ID` will be translated according to this definition.

5.2 High Level C Constructs

Since PRAM C is a C extension, many high-level C constructs are used in the functions directly or indirectly called through the `PRAM_do(K):f(...)`. This section lists those that will be translated as well as those that will not be handled.

- Function
- Expression
  - Shared variable
  - Local variable
  - Pointer
- Flow control
  - Loops
  - Conditionals

Because this report is intended to describe an algorithm for a compiler translation phase (as opposed to a design document), we do not address all corner cases of ANSI C and the extensions that comprise BEC and PRAM C. Thorough discussion of the following constructs requires understanding BEC runtime issues, a topic that is beyond the scope of this paper. At this time we do not consider the following:

- I/O statements
- MPI calls within PRAM C context
- Multiple BEC threads of execution

5.3 VP_info

In our scheme to translate `PRAM_do(K):f(...)` into BEC, each physical processor simulates the work of multiple virtual PRAM processors. In the simulation, the executions of each statement of $f(...)$ by all the virtual processors are simulated together. For example, a simple statement is translated into a loop containing the statement, with the loop count being the number of virtual processors to be simulated. In PRAM C, each virtual processor executing an instance of function $f(...)$ has its own local data, i.e. the local variables in $f(...)$. To simulate multiple virtual processors together, our approach is to replicate the local variables. This way, a local variable will be translated into a vector (referred to as generated local vector) of size equal to the number of virtual processors to be simulated on one physical processor.

A virtual processor in simulation should use its own local data, not some other virtual processor's local data. Since this local data is now stored in a generated local vector, it is necessary to have a mechanism to remember the ownership relation between a virtual processor in simulation and the index location of its local values in the generated local vectors. In a PRAM C program, some virtual processors may behave differently in $f(...)$. For example, at an if-then-else statement, some virtual processors may take the if-branch while others take the else-branch; and in a while-loop, some virtual processors may exit the loop earlier than others. It is also possible that other functions are called within $f(...)$, directly and indirectly. For correctness, the ownership mechanism needs to work in these situations.

In a PRAM C program, the special tag `MY_ID` is used inside $f(...)$ to allow a virtual processor to take special actions accordingly to its own ranking in the group of participating virtual processors in `PRAM_do(K):f(...)`. (Such a style is often used in PRAM algorithms.)

To address the above needs, we use the following structure to keep track of ownership information of a set of virtual processors to be simulated.

```
VP_info{
    int count;    // the number of active virtual processors being simulated
```

```
    int * VPIDs;  // the MY_ID values of the active virtual processors
    int * activeIndex; // activeIndex[i] remembers the index location
                       // in the generated local vectors for virtual
                       // processor VPIDs[i].
}
```

In summary, accesses to generated vectors need ownership information in `VP_info` in order to ensure that the simulation of a virtual processor uses only local data which it owns. As we shall discuss later, `VP_info` structures are generated and actively maintained in translated code.

5.4 *PRAM_do* and Initialization of VP_info

As discussed in [2], a group of K virtual PRAM processors is introduced at the `PRAM_do(K):f(...)` to execute function $f(...)$ in parallel. Each of the PRAM processors has an identification, `MY_ID`, which is the relative rank of the virtual processor within the group; so `MY_ID` is a value in the range of $[0, K-1]$. Let $P = PROC\_COUNT()$ return the number of physical processors; and let $PROC\_ID()$ return the ID of the physical processor. The idea is to let physical processor p simulate $K/PROC\_COUNT()$ virtual PRAM processors with `MY_ID` ranging from $(p-1)*(K/P)$ to $p*(K/P)-1$ (a balanced work load). The translated code in SPMD style will be

```
// declare VP_info
VP_info *VPs = malloc(sizeof(VP_info));

// set up "VPs"
VPs->count = K / PROC_COUNT();
VPs->VPIDs = malloc(VPs->count * sizeof(int));
for (int i = 0; i < VPs->count; i++) {
    VPs->VPIDs[i] = i + PROC_ID() * VPs->count;
    VPs->activeIndex[i] = i;  // All virtual processors initially active
}
// Now call the translated PRAM C function
temp_f(VPs, ...);
```

5.5 Function

In translation, a function definition (i.e., the body of the function) is either available or unavailable (such as library functions $sin(x)$ and $cos(x)$). If the function definition is available, that function body will be translated; in this case, a call to the function executed by a virtual processor does not need to be replicated in the translated code. Instead, the statements in the body of the function will be replicated in the translation. On the other hand, if the function definition is not available, a call to the function executed by a virtual processor needs to be replicated as many times as the number of the virtual processors. The key word, `PRAM_func` is required in the function declaration to indicate the function body is available for translation, either in the same file or in a separate file. For example,

```
PRAM_func extern f(int A);
```

When a function definition is available for translation, the signature, definition, and calls to that function will be translated; and the translated function will be renamed by adding prefix "temp_" to the name. The reason for this renaming is that the function may be called in both sequential and parallel context. The translation should only affect the parallel context.

5.5.1 Function Signature

When a function definition is available for translation, its function signature is translated as follows.

- **Function name** will be prepended "temp_".
- **Return type** of the translated function will be *void*. If the original return type is of data type T (not void), an additional argument will be added as follows.

  ```
  T * temp_returnValue
  ```

- **A non-shared formal argument** is translated to a new argument with one additional dimension (by adding [] in C). The reason that non-shared argument types are turned into arrays is that each virtual processor calling the function should be able to pass in different values. For example, argument *int A* will be translated into *int temp_A*[]. Such generated vectors will be referred to as generated formal argument vectors.

- **A shared formal argument** will stay the same.
- An additional argument will be added as follows.

```
VP_info *VPs;
```

For example, a function with the signature

```
T1 foo ( T2 a, shared T3 b, ... )
```

is rewritten to:

```
void foo ( VP_info* VPs, T1 * temp_returnValue, T2* a,
           shared_ref_t b ... ).
```

5.5.2 Function call

A call to a function with translated definition will not be replicated. A call to a function without translated function definition will be replicated as many times as the number of the virtual processors to be simulated.

5.5.3 Actual Arguments

The C language only has pass-by-value arguments. The effect of pass-by-reference is achieved by passing an address (pointer value).

Assume in the PRAM C source code, function $f1(...)$ calls function $f2(...)$; and let $temp\_f1(...)$ and $temp\_f2(...)$ be their translated version, respectively.

In our translation scheme, shared formal arguments remain the same; For each non-shared formal argument of $f2(...)$, a temporary local vector is generated in the body of $temp\_f1(...)$. In the $temp\_f1(...)$, the original call to $f2(...)$ becomes a call to $temp\_f2(...)$; and the original actual argument is replaced by this generated local vector. Additional code is also inserted in $temp\_f1(...)$ for this generated local vector to capture the values of the corresponding actual argument at the function call, for all the virtual processors being simulated. (We shall refer to this vector as the generated actual argument vector.)

Furthermore, a VP_info is passed into $temp\_f2(...)$ to provide ownership information for accesses of such generated argument vectors inside the body of $temp\_f2(...)$.

5.5.4 Scope of VP_info

A VP_info is created at the entry of each function body to provide ownership information for accesses to the generated local vectors at the top level of the function body. In order to provide accurate ownership information in a compact form, a different VP_info may be constructed when the flow of control enters a conditional branch. For example, at a conditional statement or in a loop, some of the virtual processors may take the true-branch while others take the false-branch. Inside each branch, we only need to simulate exactly those virtual processors that take the branch; therefore, the new VP_info for the branch records the ownership information for only those virtual processors that are active in the branch. Once the control flow exits the branch, the VP_info used before entering the branch can continue to serve. Furthermore, the VP_info passed in as a parameter to the current function needs to be compressed similarly and maintained dynamically to reflect the nesting conditional branch level.

5.5.5 Function Body Translation

Statements in the body of a function will be translated, as discussed in detail in other sections of the paper. One important thing to point out is that the VP_info structures created inside the body of the current function are used to provide ownership information for accesses of the generated local vectors at this function body. However, the VP_info structure passed in from the caller should be used to provide ownership information for accesses to generated argument vectors.

In addition, the following will be done.

- A new label, **temp_end_label** will be added towards the end of the function body. This label precedes a group of compiler generated statements to free up some dynamically allocated space.
- **return** statements without returned values will be rewritten to

```
goto temp_end_label;
```

- **return C**, where C is a constant, will be rewritten to

```
for (i = 0; i < VP->count; i++) {
    translated_returnValue[i] = C;
}
```

- **return v**, where v is a local variable, will be rewritten to

```
for (i = 0; i < VPs->count; i++) {
    translated_returnValue[i] = temp_v[i];
}
```

  where *temp_v* is the compiler generated temporary variable for v. In this case, the following declaration will be inserted at the beginning of the translated function body.

```
T * temp_v = malloc(VPs->count * sizeof(T));
             // assuming T is the type for variable v
```

- **return Exp**, where *Exp* is an expression will first be converted to

```
v_temp = Exp;  return v_temp;
```

  for some generated variable *v_temp*; then the return statement can be translated using the scheme discussed above.

5.5.6 Local variable declarations

Each declaration is rewritten to add one extra dimension. That is, a scalar becomes a 1D array; a 1D array becomes a 2D array; and so on. The size of the additional dimension is the same as the number of the virtual processors to execute this function.

For example,

```
int i;
```

is rewritten to

```
int* i = malloc(sizeof(int) * VPs->count);
```

A **free** statement is inserted at the close of the function to release the space. If any variable has initializing expressions, it is initialized after all variables are declared. If an expression is used to initialize a variable, that expression must be translated before the assignment to the variable occurs. The details of this are explained next, in the section on expression translation.

5.6 Expressions

Translation of expressions is done by walking the parse tree in the compiler. The order of evaluation for the expression is already reflected in the structure of the parse tree. In general, the operators of the expression will not change; but the operands will be translated.

5.6.1 MY_ID

`MY_ID` will be rewritten to `VPs->VPIDs[temp_i]`, where VPs is the `VP_info` struct in the context, and *temp_i* the loop control variable for the generated enclosing loop in the translated code.

5.6.2 Function Call

If the function is not declared as a `PRAM_func`, the function call will be replicated. For example, assuming x and y are local variables.

```
y = cos(x) + sin(x);
```

will be translated into

```
float temp_x = malloc(VPs->count * sizeof(float));
float temp_y = malloc(VPs->count * sizeof(float));

// Assign values to temp_x
...

// Calculate temp_y
for (int i = 0; i < VPs->count; i++) {
   temp_y[i] = cos(temp_x[i]) + sin(temp_x[i]);
}
```

If the function is declared as a `PRAM_func`, the function will NOT be replicated. For example, assuming a, x and y are local variables.

```
y = a + func(x);
```

will be translated into

```
float temp_x = malloc(VPs->count * sizeof(float));
float temp_y = malloc(VPs->count * sizeof(float));
float temp_func = malloc(VPs->count * sizeof(float));
float temp_a = malloc(VPs->count * sizeof(float));

// call the translated function, put returned value in "temp_func"
translated_f(VPs, temp_func, temp_x);

for (int i = 0; i < VPs->count; i++) {
   j = VPs->activeIndex[i];
   temp_y[j] = temp_a[j]  + temp_func[j];
}
```

5.6.3 Shared Variables

Shared variables in expressions need not change in the translated code. However, reads of shared variables in an expression must be prefetched, via *BEC_request*() and *BEC_exchange*(), before the shared value can be used in the computation of the expression. Therefore, *BEC_request*() and *BEC_exchange*() need to be inserted before the evaluation of the expression in the generated code.

For optimization, some of the reads to different shared variables can be prefetched together, so long as it is semantically legal to do. Before the final evaluation of the expression, a shared variable write in that expression needs to be handled, by inserting a call to *BEC_exchange*(). The exchange is necessary due to possible side-effects on the shared variable.

Definition 5.1 *An* **X-block** *is defined as a maximal abstract syntax subtree that does not contain any writes into shared variables. A* **synchronization block** *is defined as an X-block, a write to a shared variable, or a function call.*

Reads from the shared variables in an X-block can be prefetched together. This is done by inserting calls to *BEC_request*(), one call per shared variable read, which are followed by only one single call to *BEC_exchange*() to get the values.

During translation, an expression is broken up into synchronization blocks. A temporary variable (dynamically allocated vector) is declared for each synchronization

block to store its intermediate values. These values are put together after code has been emitted for all synchronization blocks that contribute to the final value of the expression.

Note: For future optimizations, more sophisticated analysis could allow the compiler

- to determine when shared variable reads and writes can be served in the same $BEC\_request()$; and
- in general, to combine independent variable accesses in the same synchronization block.

Execution within a local block by different virtual PRAM processors does not require synchronization. Their execution can proceed on each processor separately; and intermediate values can be stored in the variables declared for them. For example, translation of $a + b$, where a is a local variable and b is a formal argument, is shown below. Let VPs be the `VP_info` structure created in the current function and valid at the point of the evaluation of this expression; and let arg_VPs be the `VP_info` structure passed as a argument from the caller.

```
// declaration block
...

// note that declaration for a and b are translated to their
// respective array types, as explained previously.
int* temp_a = malloc(sizeof(int)*VPs->count);

// here is the temporary variable to store the
// intermediate value of a + b
int* temp_a_plus_b = malloc(sizeof(int)*VPs->count);

// Other code
...

// translation for a+b
for (i=0; i<VPs->count; i++) {
    temp_a_plus_b[VPs->activeIndex[i]] =
           temp_a[VPs->activeIndex[i]] +
           temp_b[arg_VPs->activeIndex[i]];
}
```

```
// free statements to release the space
free(temp_a);
...
```

Execution of each X-block by different virtual PRAM processors needs to be synchronized (Note: A function that refers only to local variables does not need to be synchronized. However, since this might be a function linked in from previously compiled files and the compiler cannot know whether it involves shared references, we choose to synchronize all function executions.) For functions calls, this translation is fairly simple: we call the function, which has been translated to take the appropriate arguments: a `VP_info` indicating the virtual processors to simulate, and the arrays for arguments and return value.

For shared variable accesses, however, we need to generate the following calls to the BEC runtime library: *BEC_Request* (for read or write), *BEC_Exchange* (signaling this processor is halting and ready for exchange of data), and *BEC_Read* (if this is a read request). For example, for shared variable c and local variable i, expression $c[i]$ is translated to:

```
// Declare intermediate variable to store the values.
int* temp_c_sub_i = malloc(sizeof(int)*VPs->count);
int* temp_i = malloc(sizeof(int)*VPs->count);
// Issue BEC runtime library request.
for (j=0; j<VPs->count; j++) {
   k = VPs->activeIndex[j];
   BEC_Request(c, temp_i[k], 1);
}

// Issue exchange request.
BEC_Exchange();

// When exchange returns, all shared data is ready. Read data.
for (j=0; j<VPs->count; j++) {
   k = VPs->activeIndex[j];
   BEC_Read(c, temp_i[k], 1, &(temp_c_sub_i[k]));
}
```

As apparent in the example above, a physical processor can now "bundle" all the read requests for multiple virtual processors and issue them at the same time. Thus PRAM C exploits the parallelism in communication, and saves communication overhead.

Now consider another example, $a + b + c[i]$, where a is a local variable, b is a formal argument, and c is a shared array. Let VPs be the `VP_info` structure created in the current function and valid at the point of the evaluation of this expression; and let arg_VPs be the `VP_info` structure passed as a argument from the caller. In the translated code, we combine the variable declarations for the two code examples above, issue the code block generated for $a + b$ first, then the code block for $c[i]$, and finally,

```
// declare a variable to hold a+b+c[i]
int* temp_a_plus_b_plus_c_sub_i = malloc(sizeof(int)*VPs->count);

// code generated for the two blocks, putting everything together:
for(i=0; i<VPs->count; i++) {
    j = VPs->activeIndex[i];
    temp_a_plus_b_plus_c_sub_i[j] =
        temp_a_plus_b[j] + temp_c_sub_i[j];
}
```

We could potentially generate lots of local variables to keep track of intermediate values for expressions. This situation could be alleviated with more static analysis by the compiler, where certain expressions do not need local variables for their values. For example, for a statement expression, where the expression is executed merely for its side effect, no variable is needed to store the final value. We leave this as future optimization.

5.7 Assignment Statement

Simple statements such as ID = *expr* ; are easy to translate. We translate the expression *expr* according to the rules described above.

Compound statements involve loops (i.e. *while*), branching (i.e. *if-then-else*), and jumps (i.e. *goto*, *break*). Extra care is needed to ensure preservation of the program semantics for all virtual processors being simulated. Here we briefly describe how such situations are handled.

5.8 Loops

The C language has several loop constructs such as **while-loop** and **for-loop**. Their translation schemes are similar. In this section, we present the algorithm for translating the while-loop, just to show the idea.

```
while (condition) {
    body;
}
```

To simulate the virtual PRAM processors executing a while loop, the work is assigned to each of the physical processors. Even if the workload is balanced in terms of the number of virtual processors to be simulated on each physical processor, it is still possible that some of the physical processors finish the while-loop simulation much earlier than others. There is a potential problem in that an active physical processor may call a *BEC_exchange*(). Since *BEC_exchange*() is also a barrier, that active physical processor cannot proceed until the call is responded to by other physical processors, some of which may already be out of the loop due to the smaller number of iterations they executed. This can potentially leave that active processor hanging forever.

Fortunately, the PRAM C semantics defines an implicit barrier at the exit of a while-loop (as a compound statement). That means the processors finishing while-loop early must wait for other processors. One possible solution to the above problem is for the compiler to insert an extra busy-waiting loop at the end of generated code for the original while-loop. This extra busy-waiting loop will keep checking a "global" status flag, *some_processor_busy*, for the status of all the other physical processors about the execution of the translated while-loop code; and in the meantime, the busy-waiting loop will keep calling *BEC_exchange*(), on the shared variable *physical_processor_busy*. This provides the necessary response to other active physical processors.

Note: In the busy-waiting loop, the computation of the global status using a for-loop on vector *physical_processor_busy*[ ] can be more efficiently implemented using a global reduction collective. But since this is a busy-waiting loop, performance is not too important here.

The translation algorithm for a **while-loop** follows. Here *emit*(*str*) is a function to dump *str* into the stream of generated code.

```
void translate_while_loop(... , Statement * while_loop) {
    // Input assumption:
    //     - VPs: the VP_info created in the current function
    //       and is valid at the (top of)  while loop
    //     - arg_VPs: the VP_info is the formal argument in the
    //       translated function containing this translated while loop

    // emit a block of code
    emit(
          // status flag
          int some_processor_busy = 1;
          // declare a shared vector to keep track of the status
          // of the physical processors
          shared int * physical_processor_busy =
                       BEC_joint_malloc(PROC_COUNT() * sizeof(int));

          // This physical processor sets its own status to "not done"
          physical_processor_busy[PROC_ID] = 1;

          // declare "VPs_true" to be the compressed VP_info to hold
          // the local vector ownership information of those VPs
          // for which "condition" is true
          VP_info *VPs_true = malloc(sizeof(VP_info));
          VPs_true->VPIDs = malloc(sizeof(int)*VPs->count);
          VPs_true->activeIndex = malloc(sizeof(int)*VPs->count);

          // declare "arg_VPs_true" to be the compressed VP_info to hold
          // the argument vector ownership information of those VPs
          // for which "condition" is true
          VP_info *arg_VPs_true = malloc(sizeof(VP_info));
          arg_VPs_true->VPIDs = malloc(sizeof(int)*VPs->count);
          arg_VPs_true->activeIndex = malloc(sizeof(int)*VPs->count);

          // declare variable to hold value of condition expression.
          int* temp_cond = malloc(sizeof(int)*VPs->count);
    );

    // emit code to compute initial "temp_cond" array for all VPs
    translate_condition(VPs, while_loop->condition, "temp_cond");

    emit(
          // Construct the compressed "VPs_true" based on "VPs"
          VPs_true->count = 0;
```

```
      for (i=0; i<VPs->count; i++) {
         if ( temp_cond[i] ) {
             // add VPs->VPIDs[i] to list of true VPs
             VPs_true->VPIDs[VPs_true->count++] = VPs->VPIDs[i];
             VPs_true->activeIndex[VPs_true->count] =
                                              VPs->activeIndex[i];
         }
      }
      // Similarly, construct the compressed "arg_VPs_true"
      // based on "arg_VPs"
      ...
);

emit(
      while(VPs_true->count > 0) {

         // translated the body of the while_loop
         translate(VPs_true, while_loop->body);

         // clear the list of true VPs
         VPs_true->count = 0;
);

// emit code: to recalculate the temp_cond array --
// only for the VPs already in the loop
translate_condition(VPs_true, while_loop->condition,
                                          "temp_cond");

emit(
      // Construct the compressed "VPs_true" based on "VPs"
      for (i=0; i<VPs->count; i++) {
         if ( temp_cond[i] ) {
             // add VPs->VPIDs[i] to list of true VPs
             VPs_true->VPIDs[VPs_true->count++] = VPs->VPIDs[i];
             VPs_true->activeIndex[VPs_true->count] =
                                              VPs->activeIndex[i];
         }
      }

      // Similarly, construct the compressed "arg_VPs_true"
      // based on "arg_VPs"
      ...
```

```
        } // end while
    );

    emit(
        // This physical processor sets its own status to "done"
        physical_processor_busy[PROC_ID] = 0;

        // the extra busy waiting loop
        // (until all other processors are done)
           while (some_processor_busy) {
               // Read the shared vector "physical_processor_busy"
               BEC_request(physical_processor_busy, 0, PROC_COUNT());
               BEC_exchange();

               // Combine the global status
               for (int i = 0; i < PROC_COUNT(); i++) {
                   some_processor_busy |= physical_processor_busy[i];
               }
           }
     );

    // emit code to free up malloc() space
    ...
}
```

Note: The generated code of a compound statement such as the while-loop will be placed in a C block statement enclosed within curly braces "{" and "}". This provides the appropriate scope for the generated temporary `VP_info` structures (declared within the block statement), such as *VPs_true*, in order to avoid name collision when multiple compound statements need to be translated. (The same applies to the translation of if-then-else as presented in the next section.)

5.9 Conditional Statements

The C language has conditional statements such as **if-then-else**, **switch-case**, etc. Their translation schemes are similar. Here we present only the translation algorithm for **if-then-else**.

```
if (condition) {
    if_stmts;
} else {
    else_stmts;
}
```

The PRAM C semantics specifies that all the *if-stmts* must synchronize across all (virtual) processors executing them. The same is true for the *else-stmts*. However, the two sets of processors for these different branches do not have to synchronize at all. Therefore, in our translation, we can translate the if-branch first, and then the else-branch, while maintaining the correct semantics. This translation algorithm is as follows.

Similar to the problem in the while-loop translation, it is possible that on one physical processor, all the virtual processors simulated take only the if-branch; while on another physical processor, all the virtual processors simulated take only the else-branch. Besides, in the generated code, the number of calls to *BEC_exchange*() on the two branches may be different. This difference can potentially leave some physical processors hanging on the call to *BEC_exchange*(), waiting for other physical processors' responses. Once again, one possible solution is to use the same busy-waiting loop as in the translated code for the while-loop to resolve this issue. Specifically, we insert two such extra busy-waiting loops, one after the translated code for the if-branch, and the other after the translated code the else-branch.

The translation algorithm for **if-then-else** is as follows.

```
void translate_if_then_else(..., Statement * if_then_else) {
    // Input assumption:
    //     - VPs: the VP_info created in the current function
    //       and is valid before the conditional statement
    //     - arg_VPs: the VP_info is the formal argument in the
    //       translated function containing this conditional

    emit(
        // status flag
        int some_processor_busy = 1;

        // declare a shared vector to keep track of the status
        // of the physical processors
        shared int * physical_processor_busy =
                    BEC_joint_malloc(PROC_COUNT() * sizeof(int));
```

```
      // This physical processor sets its own status to "not done"
      physical_processor_busy[PROC_ID] = 1;

      // declare variable to hold VPs for which "condition" is true
      VP_info *VPs_true = malloc(sizeof(VP_info));
      VPs_true->VPIDs = malloc(sizeof(int)*VPs->count);
      VPs_true->activeIndex = malloc(sizeof(int)*VPs->count);

      // declare variable to hold VPs for which "condition" is false
      VP_info *VPs_false = malloc(sizeof(VP_info));
      VPs_false->VPIDs = malloc(sizeof(int)*VPs->count);
      VPs_false->activeIndex = malloc(sizeof(int)*VPs->count);

      // Similarly, declare "arg_VPs_true" and "arg_VPs_false"
      // to be the compressed VP_info structures to hold
      // the argument vector ownership information of those VPs
      // in the if-branch and the else-branch, respectively.
       VP_info *arg_VPs_true = malloc(sizeof(VP_info));
       arg_VPs_true->VPIDs = malloc(sizeof(int)*VPs->count);
       arg_VPs_true->activeIndex = malloc(sizeof(int)*VPs->count);

       VP_info *arg_VPs_false = malloc(sizeof(VP_info));
       arg_VPs_false->VPIDs = malloc(sizeof(int)*VPs->count);
       arg_VPs_false->activeIndex = malloc(sizeof(int)*VPs->count);

      // declare variable to hold value of condition expression.
      int* temp_cond = malloc(sizeof(int)*VPs->count);
 );

      // emit code to compute initial "temp_cond" array for all VPs
      translate_condition(VPs, if_then_else->condition, "temp_cond");

emit(
     // collect all the VPs for which temp_cond is true
     // into array trueVPs, others into falseVPs
     VPs_true->count = 0;
     VPs_false->count = 0;
     for (i=0; i<VPs->count; i++) {
        if ( temp_cond[i] ) {
            // add VPs->VPIDs[i] to list of true VPs
            VPs_true->VPIDs[VPs_true->count++] = VPs->VPIDs[i];
            VPs_true->activeIndex[VPs_true->count] =
```

```
                                                VPs->activeIndex[i];
        } else {
            VPs_false->VPIDs[VPs_false->count++] = VPs->VPIDs[i];
            VPs_false->activeIndex[VPs_false->count] =
                                                VPs->activeIndex[i];
        }
    }

    // Similarly, construct the compressed "arg_VPs_true"
    // and "arg_VPs_false" based on "arg_VPs"
    ...

    // translate the if branch.
    if (VPs_true->count > 0) {
);

// translate the if_stmts only for the true_VPs
translate(VPs_true, if_stmts);

emit(
      // This physical processor sets its own status to "done"
      physical_processor_busy[PROC_ID] = 0;

      // the extra busy waiting loop
      // (until all other processors are done)
      while (some_processor_busy) {
          // Read the shared vector "physical_processor_busy"
          BEC_request(physical_processor_busy, 0, PROC_COUNT());
          BEC_exchange();

          // Combine the global status
          for (int i = 0; i < PROC_COUNT(); i++) {
             some_processor_busy |= physical_processor_busy[i];
          }
      } // end while
  );

emit(
      } // end if

      // translate the else branch.
      if (VPs_false->count > 0) {
```

```
            some_processor_busy = 1;
            // This physical processor sets its own status to "done"
            physical_processor_busy[PROC_ID] = 0;
);

// translate the else_stmts only for the false VPs
translate(VPs_false, if_stmts);

emit(
      // This physical processor sets its own status to "done"
      physical_processor_busy[PROC_ID] = 0;

      // the extra busy waiting loop
      // (until all other processors are done)
      while (some_processor_busy) {
         // Read the shared vector "physical_processor_busy"
         BEC_request(physical_processor_busy, 0, PROC_COUNT());
         BEC_exchange();

         // Combine the global status
         for (int i = 0; i < PROC_COUNT(); i++) {
            some_processor_busy |= physical_processor_busy[i];
         }
      }
 );

emit(
     } // end if
);

// emit code to free up malloc() space
...
```

5.10 Other Flow Control Statements

- `break`: *break* statements can be handled easily with our scheme of translating loops. When a *break* statement is issued for a particular virtual processor, we simply remove the virtual processor's ID from the VPs_true list. Since recalculation of conditions applies only to those virtual processors already in the list, the exited processor will not be considered for the loop again.

- `continue`: *continue* statements can be rewritten with *if* statements.
- `goto`: *goto* statements are not allowed in PRAM C ([2]).

6 Translating UPC into BEC

In its most general form, the UPC parallel loop construct takes the form

```
upc_forall(e1; e2; e3; e4) {
   body;
}
```

The expressions `e1; e2; e3` are used to control iterations of the loop, while `e4`, called an affinity expression in UPC, suggests which physical processor to use for a given iteration. We shall leave this most general form for future discussion.

For now, we will focus on a common form of the `upc_forall` construct in UPC programs, the counter loop. The loop control expressions can be quite complex; however, standard compiler techniques can normalize general counter loops to a more specialized form. Therefore, the following example suffices to illustrate the translation of a UPC parallel (counter) loop to BEC.

```
upc_forall(i = 0; i < N; i++; affinity_expression) {
   body;
}
```

Note: There is no data dependence permitted between different iterations of the UPC loop [8]; therefore, different iterations can be executed in arbitrary relative order.

6.1 UPC to BEC Translation Scheme

To see the general idea of the translation scheme, we break the translation process into small, comprehensible steps. For ease of exposition, we give the affinity expression a definite form. Assume that array $A[N]$ is a shared variable that is accessed in the loop body.

```
upc_forall(i = 0; i < N; i++; &A[i]) {
   body;
}
```

Here the intent of the affinity expression is a hint that iteration i should be executed by the processor that physically holds data element $A[i]$.

Assume that P is the number of physical processors available at runtime. A transformation from UPC to UPC leads to

```
upc_forall(j = 0; j < P; j++; &A[j*(N/P)]) {
   // The inner loop is a sequential for loop
   for (k = 0; k < (N/P); k++) {
       i = j * (N/P) + k;
       body;
   }
}
```

It is possible to provide explicit barriers in UPC. Assume the loop body consists of code in phases, where each phase is some C code followed by a barrier. That is, a phase further consists of

```
code;
barrier;
```

Therefore, the original `upc_forall` loop becomes

```
upc_forall(j = 0; j < P; j++; &A[j*(N/P)]) {
   // The inner loop is a sequential for loop
   for (k = 0; k < (N/P); k++) {
       i = j * (N/P) + k;
       code_1;
       barrier;
       ...
       code_last;
       barrier;
   }
}
```

Note that `code_x` is sequential C code with no virtual processors (in the PRAM C sense). The generated BEC code will be as follows.

```
for (k = 0; k < (N/P); k++) {
    i = PROC_ID() * (N/P) + k;

    [code_1 translated]
    ...
    [code_last translated]
}
// Exchange to take care of the last batch of shared variable writes
BEC_exchange();
```

Here we use `[code_x translated]` to represent the translation of UPC `code_x` into BEC. The enclosing `upc_forall` loop is no longer necessary because BEC is SPMD. The UPC explicit barriers are naturally subsumed by the BEC data exchange semantics.

Note: The translation ignores the affinity expression because the BEC environment manages data locality of shared variables.

The translation of each UPC code phase can be done as follows.

- Keep the original `code_x` (as C code) unchanged; and insert the following two pieces before it;
  - Since a `BEC_request` is needed for every shared variable read access, an easy way to do so is to replicate the control constructs of the code, and then to replace the simple statements in code with `BEC_request` calls, one for each shared variable read in the statements.
  - Insert a `BEC_exchange` call after the group of replicated and modified statements.

The translation of UPC to BEC appears to be more straightforward than that of PRAM C to BEC. This apparent simplicity may be due to different ways of handling "virtual processors" and local data in the two high level languages.

7 Comparison of PRAM C and UPC

In this section, we address differences and similarities of translating the PRAM_do construct and the upc_forall construct. For this purpose, it is important to examine the semantics of these two parallel constructs.

Consider the PRAM_do in its general form,

```
// PRAM C construct
PRAM_do(N): f(...);
```

and the upc_forall loop in its commonly used form as a counter loop.

```
// UPC construct
upc_forall(i = 0; i < N; i++; affinity_expression)
    body;
```

Note: If the affinity expression is "continue" (not integer or pointer to shared) or if it is not specified, the upc_forall is reduced to a sequential for-loop.

7.1 Observations

Similarities:

- Both constructs are used to express parallelism, up to N parallel threads of execution. In PRAM_do, N instances of function $f(...)$ will be executed, possibly in parallel; while in upc_forall, N instances of *body* will be executed.

Differences:

- **Nesting**
  - Nested upc_forall constructs are reduced to sequential for-loops.
  - Nesting of PRAM_do is allowed and the parallel semantics of the nested PRAM_do is unchanged.
- **Programming in virtual processors**
  - PRAM_do enables expression of parallelism in virtual processors. That is, each instance of the function $f(...)$ will be executed by one virtual processor.
  - In upc_forall, each instance of *body* somewhat resembles an instance of function $f(...)$ in PRAM_do. But they are different.

* In PRAM_do, each instance of $f(...)$ can have its own local variables, which are truly private to the executing virtual processors. (Communication between different instances of $f(...)$ can only be through shared variables or shared arguments.)
* Local variables in UPC pertain to physical processors. Therefore, local variables in UPC are potentially shared by multiple instances of the executing *body* assigned to run on the hosting physical processor. So local variables in UPC are not private to one instance of the body, but are really shared by a group of instances. In this sense, using upc_forall does not fully support expression of parallelism in virtual processors.

- **Work assignment to physical processors**
  - Depending on the affinity expression in ups_forall, the group of instances assigned to a physical processor needs to be determined at compile time (and even at runtime). Consequently, users need to ensure that sharing of local variables by multiple *body* instances does not lead to unexpected program behavior. Explicit synchronizations (e.g., using barriers) may be needed for program correctness; but such synchronization points apply to all the instances on all physical processors globally. Currently, there is no synchronization mechanism in UPC to deal with such sharing of local variables on each physical processor. (This complication may be undesirable for the ease of programming.)
  - PRAM_do does not have such an issue since all the local variables are truly private to the virtual processors. Work assignment to physical processor can either be done at compile time or runtime, beyond the concern of the user.

- **Difference in Semantics**
  - Statements in $f(...)$ of PRAM_do follow implicit barriers ([2]); so the parallel execution of the instances of $f(...)$ is synchronous.
  - Statements in a upc_forall *body* follow only explicit barriers ([8]), so the parallel execution of the instances of *body* is asynchronous.

7.2 Differences in Translations

The main difference in translating PRAM_do and translating upc_forall is the different treatment of local variables. In translating PRAM_do, local variables in $f(...)$ must

be replicated; but in translating upc_forall, local variables need not change.

Implicit barriers in PRAM C do not lead to additional difficulty in translation, because they can be treated as explicit barriers at predetermined locations.

8 Conclusion

In this report we have shown that two GAS languages, UPC and PRAM C, can be compiled into BEC. This shows that BEC can be used an intermediate GAS language for higher level languages. As it was discussed in this report, BEC consists of a simple C extension and a runtime library. The BEC runtime library is just a little more than a simplified and formalized common API to some of the current utilities such as the EPETRA package in Trilinos [4]. Therefore, any platform-specific optimizations that have been made to those utilities are available for use by the BEC runtime.

For implementations of high level GAS models, this provides a new and alternative approach which can leverage many of the existing infrastructures on current platforms. This is in contrast to some of the existing UPC implementations such as the Berkeley UPC [3], which develops its capabilities by reimplementation of distributed message-passing functions.

As an alternative to the thread-based implementation PRAM C for the virtual processors described in [2], this translation approach is amenable to compiler optimizations. Topics such as prefetching of remote data, interprocedural optimization of data access, and data exchange heuristics are some of the areas we plan to explore in the future.

The theoretical significance of our translation approach is that it bridges between fine-grained parallelism, as in PRAM algorithms, and coarse-grained communication, as in the BSP model.

References

[1] Jonathan L. Brown, Sue Goudy, Mike Heroux, Shan Shan Huang, and Zhaofang Wen. BEC: A virtual shared memory parallel programming environment. Technical Report in preparation, Sandia National Laboratories, Albuquerque, NM, 2005.

[2] Jonathan L. Brown and Zhaofang Wen. PRAM C: A new parallel programming environment for fine-grained and coarse-grained parallelism. Technical Report SAND2004-6171, Sandia National Laboratories, 2004.

[3] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS 2003)*, 2003.

[4] Michael A. Heroux. Trilinos home page. http://software.sandia.gov/trilinos, 2004.

[5] J. JaJa. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[6] NPACI. SHMEM tutorial page. http://www.npaci.edu/T3E/shmem.html, 2005.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1, The MPI core.* The MIT Press, 1998.

[8] The UPC Consortium. *UPC Language Specification*, 2005.

[9] Berkeley UPC Working Group (url). Berkeley UPC home page. http://upc.lbl.gov, 2005.

[10] Leslie G. Valiant. A bridging model for parallel computation. *Comm. ACM*, August 1990.

[11] E. Wiebel, D. Greenberg, and S. Seidel. UPC collective operations specification. 2003.

This page intentionally left blank

DISTRIBUTION:

1 MS 1110
John Aidun, 1435

1 MS 0817
James Ang, 1422

1 MS 0807
Bob Ballance, 4328

1 MS 0817
Bob Benner, 1422

1 MS 0376
Ted Blacker, 1421

1 MS 1110
Ron Brightwell, 1423

1 MS 1110
Jonathan L. Brown, 1423

1 MS 0382
Kevin Brown, 1543

1 MS 0320
William J. Camp, 1400

1 MS 1110
S. Scott Collis, 1414

1 MS 0318
George Davidson, 1412

1 MS 1110
Erik DeBenedictis, 1423

1 MS 0817
Doug Doerfler, 1422

1 MS 0316
Sudip Dosanjh, 1420

1 MS 0817
Brice Fisher, 1422

1 MS 0382
Mike Glass, 1541

1 MS 0817
Sue Goudy, 1422

1 MS 8960
James Handrock, 9151

1 MS 1110
William Hart, 1415

1 MS 0822
Rena Haynes, 1424

1 MS 1110
Bruce Hendrickson, 1414

1 MS 1110
Michael Heroux, 1414

1 MS 0316
Scott Hutchinson, 1437

1 MS 0817
Sue Kelly, 1422

1 MS 0378
Marlin Kipp, 1431

1 MS 1111
Patrick Knupp, 1411

1 MS 0801
Rob Leland, 4300

1 MS 0370
Scott Mitchell, 1411

1 MS 1110
Steve Plimpton, 1412

1 MS 0807
Mahesh Rajan, 4328

1 MS 1110
Rolf Riesen, 1423

1 MS 0378
Allen Robinson, 1431

1 MS 0318
Elebeorba May, 1412

1 MS 0321
Jennifer Nelson, 1430

1 MS 1110
Cynthia Phillips, 1415

1 MS 1110
Neil Pundit, 1423

1 MS 1111
Mark D. Rintoul, 1412

1 MS 1110
Suzanne Rountree, 1415

1 MS 1111
Andrew Salinger, 1416

1 MS 0378
Stewart Silling, 1431

1 MS 0378
James Strickland, 1433

1 MS 0378
Randall Summers, 1431

1 MS 1110
Jim Tomkins, 1420

1 MS 0370
Tim Trucano, 1411

1 MS 0817
John VanDyke, 1423

1 MS 0817
Courtenay Vaughan, 1422

1 MS 1110
Zhaofang Wen, 1423

1 MS 0822
David White, 1424

1 MS 1110
David Womble, 1410

1 MS 0823
John Zepper, 4320

2 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616

Second Printing, (March 2006):

1 Dr. Stan Ahalt
205 Dreese Laboratory
Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210 USA

1 Dr. David Bailey
Lawrence Berkeley National Laboratory
Mail Stop 50B-2239
Berkeley, CA 94720

1 Dr. Dan Bonachea
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Bill Carlson
IDA Center for Computing Sciences
17100 Science Drive
Bowie, MD 20708

1 Dr. Bradford Chamberlain
Cray Inc.
411 First Avenue S, Suite 600
Seattle, WA 98104

1 Dr. Mark Davis
Intel
110 Split Brook Road - SPT 1
Nashua, NH 03062-2711

1 Dr. Jason Duell
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Tarek El-Ghazawi
Department of Electrical and Computer Engineering
The George Washington University
801 22nd Street NW 6th floor
Washington DC 20052

1 Dr. Rob Fowler
Department of Computer Science
Rice University
P.O. Box 1892, MS 132
Houston, TX 77251
USA

1 Dr. Al Geist
Oak Ridge National Laboratory
P.O. Box 2008
Bldg 6012
Oak Ridge, TN 37831-6367

1 Dr. Alan George
Department of Electrical and Computer Engineering
University of Florida
PO Box 116200
327 Larsen Hall
Gainesville, FL 32611-6200

1 Dr. Robert Graybill
DARPA
3701 Fairfax Drive
Arlington, VA 22203

1 Dr. Bill Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

1 Dr. Joseph JaJa
Institute for Advanced Computer Studies (UMIACS)
A.V. Williams Building
University of Maryland
College Park, MD 20742-3251

1 Dr. Fred Johnson
DOE
SC-31/Germantown Building
1000 Independence Avenue SW
Washington, DC 20585-1290

1 Dr. Laxmikant Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302

1 Dr. Ashok Krishnamurthy
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212

1 Dr. Vipin Kumar
Department of Computer Science and Engineering
University of Minnesota
200 Union Street SE
4-192 EE/CS
Minneapolis, MN 55455

1 Dr. Rusty Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439

1 Dr. John Mellor-Crummey
Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251
USA

1 Dr. Juan Meza
Lawrence Berkeley National Laboratory
Mail Stop 50B-2239
Berkeley, CA 94720

1 Dr. Phil Merkey Department of Computer Science
Michigan Tech University
1400 Townsend Dr.
Houghton, MI 49931

1 Dr. Robert W Numrich
Minnesota Supercomputing Institute
University of Minnesota
599 Walter Library
117 Pleasant St. SE
Minneapolis, MN 55455

1 Dr. Steve Reinhardt
SGI
2750 Blue Water Road
Eagan, MN 55121

1 Dr. Vivek Sarkar
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

1 Dr. P. Sadayappan
Department of Computer Science and Engineering
595 Dreese Lab
2015 Neil Avenue
Ohio State University, Columbus, Ohio 43210 USA

1 Dr. Steve Seidel
Department of Computer Science
Michigan Tech University
1400 Townsend Dr.
Houghton, MI 49931

1 Dr. Lauren Smith
NSA
1806 Stonegate Ave.
Corfton, MD 21114

1 Dr. Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801-2302

1 Dr. Guy Steele
Sun Microsystems
1 Network Drive
Burlington, MA 01803

1 Dr. Thomas Sterling
Center for Computation and Technology
Department of Computer Science

202 Johnston Hall
Louisiana State University
Baton Rouge, LA 70803

1 Dr. Uzi Vishkin
Institute for Advanced Computer Studies (UMIACS)
A.V. Williams Building
University of Maryland
College Park, MD 20742-3251

1 Dr. James (Trey) White III
Oak Ridge National Laboratory
1 Bethel Valley Road
PO Box 2008 MS-6008
Oak Ridge, TN 37831-6008

1 Dr. Brian Wibecan
UPC Development Team
Hewlett-Packard Company
110 Split Brook Road
ZKO1-3/D40
Nashua, NH 03062-2698

1 Dr. Kathy Yelick
777 Soda Hall
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776

1 Dr. Thomas Zacharia
Oak Ridge National Laboratory
1 Bethel Valley Road
Box 2008
Oak Ridge, TN 37831

1 Dr. Hans Zima
Principal Scientist
JPL
Caltech
4800 Oak Grove Drive MS 171-373
Pasadena, CA 91109