All Parallel Paradigms in Two Operators

Stanislav Bratanov
Stanislav.Bratanov@hotmail.com

***Abstract*—This paper introduces a new parallel programming language called C= (pronounced: 'Sea Stripes') which defines two basic operators to parallelize and serialize execution. The parallelization operator is designed to handle two major classes of parallelism – lightweight task based, and independent thread based – within the same syntactic construct. That is achieved by detecting explicit thread synchronization constructs and transparently converting tasks to threads, and vice versa. The default lightweight task semantics allows for arbitrary nesting of parallel operators without abusing system resources. A novel task management scheme (or, more correctly, task generation scheme) facilitates automatic work balancing and eliminates the memory and performance overhead typically associated with allocating task queues and work stealing. The same operator is also used to express heterogeneous parallelism and balance execution between central processors and accelerators, and we plan to extend this solution to distributed environments, as those, from our language's point of view, are similar to heterogeneous systems. The serialization operator provides for fine-grain memory access synchronization without introducing synchronization objects per access points: instead, each parallel thread uses a hash function to detect memory access collisions and self-suspends. The C= language reuses C/C++ syntax structure information, such as visibility scopes, to abstract as many parallel programming paradigms as possible with minimal syntactic overhead.**

Keywords—parallel; heterogeneous; language

I. MOTIVATION

Why does one need to add yet another item onto the list of parallel languages cited in [3]? Because they'd like to:

- simplify basic categories for parallel programming;
- find a common ground and combine multiple programming paradigms into (ultimately, as our intuition tells us) a single linguistic construct;
- stitch tasking and threading – the gap none of the modern parallel runtimes attempts to bridge;
- cover emerging programming paradigms of heterogeneous systems;
- suggest a new approach to automating distributed programming, consistent with the above, other than (or in addition to) explicit message passing and laying out memory variables over nodes;
- reuse sequential syntax and semantics to lower the programmers' burden: utilize visibility scopes to convey extra information "for free", and employ operator scope to enable definition and use of parallel logic at the same spot (rather than utilizing call-based semantics as in [6]);
- ease the penetration of parallel programming into new areas by enabling adoption of the same basic semantic concepts in domain-specific languages;
- facilitate creation of higher-level programming languages utilizing parallelism (e.g., functional languages with implicit parallelism) by providing a common platform the new languages can compile to;
- anticipate creation of joint parallel runtimes to fight resource over-utilization, by proving that the basic concepts of the language are truly common for a significant subset of the runtimes;
- and save man-hours to develop, debug, and optimize a parallel program by ensuring the same program can effectively run on the majority of parallel architectures without modification, and can be debugged on just one of them.

Further in this document Section II lists the requirements for a new parallel language. Section III describes our solution, while Section IV proves the solution to be viable by providing performance scalability and precision data for a popular benchmark.

II. REQUIREMENTS

A new parallel programming language, to achieve the goals listed in the previous section, has to fulfill the following technical requirements:

A. Nested Parallelism

Support nested parallelism (as it is common to reuse parallel libraries within an already parallel program) and avoid compute-resource over-utilization – unlike OpenMP [7], which hits resource limits pretty quickly.

B. Synchronous Parallelism

Support tasking paradigms, that is, execution of parallel work items that have to run to completion (all task queuing and popular parallel_for constructs fall into this category, e.g., [4] and [5]).

C. Asynchronous Parallelism

Cover what now has to be manually implemented using low-level system API (e.g., [12] and [13]) but constitutes a primary programming paradigm for a large class of programs (interactive/graphics/gaming/multimedia software, parallel runtimes themselves, system software), when the amount of work is not known beforehand, and the execution is triggered by external conditions.

D. Heterogeneous Parallelism

Expand computation to co-processors (e.g., GPUs) like in [9], [10], and [11], but eliminate the need to manually setup execution contexts and write separate functions, plus ensure full control over execution and enable concurrent CPU and GPU utilization.

E. Distributed Parallelism

Be suitable for work on clusters and grids or in any other network-connected environment.

F. Serialization of Parallel Execution

Define flexible arbitration constructs to synchronize access to memory objects, with finer granularity and more naturally connected with the language than OpenMP's *__critical* directive or special data types in TBB and Cilk or low-level OS synchronization primitives.

III. Solution

Our solution comprises the following two primary operators that account for the majority of parallel programming paradigms. The explanation below refers to the accompanying poster and to Fig. 1 as an example of C= code.

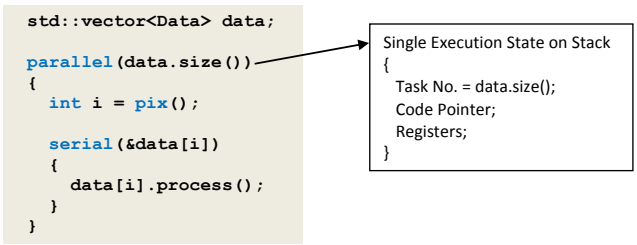


Fig. 1. C= parallel and serial operator example.

A. Parallel Operator

This operator implements what was referred to above as synchronous, asynchronous, and heterogeneous parallelism.

1) Tasking (synchronous parallelism)

We define a task as a portion of work to be executed by a thread; while a thread is a system-level execution entity (a program, even a sequential one, is always executed within a context of at least one thread). Each task is represented by an execution state, which is hardware and system-dependent and, typically, includes registers in accordance with a software calling convention.

C= *parallel* operator works as follows:

(1) The thread executing a sequential program encounters the *parallel* operator; (2) it saves the current execution state, plus computes and saves the total number of tasks that need to be formed out of the saved state and executed; (3) the thread fetches the first task (so-called primary task or thread) and signals to a pool of threads from C= runtime library to start fetching and executing other tasks; (4) each thread of the C= runtime thread pool fetches a task (by copying and modifying the saved register state) and executes it; (5) the procedure repeats until there's no task left; (6) the task execution ends at the closing brace of the *parallel* operator, so no task/thread can continue execution beyond the parallel scope, except for the primary thread that entered the *parallel* operator; (7) the primary thread continues executing the sequential program, while pooled threads wait on a semaphore.

As a result, such a scheme of parallel execution management automatically enables dynamic work balancing and efficient processor utilization, and eliminates thread conflicts due to work stealing: because each thread *forms* a new task from the saved execution context independently, rather than *stealing* the work from other threads, and the faster the thread executes, the more tasks it fetches and forms – hence dynamic balancing. Besides, one *parallel* operator introduces only one register state structure on the stack for any number of tasks, and no task queue whatsoever.

It should be also noted that C= makes logical use of the variable visibility scopes: all variables declared outside a parallel region are equally accessible from all parallel tasks or threads, while all data declared inside a parallel region are task/thread private.

2) Threading (asynchronous parallelism)

In many cases there is a need for a dedicated thread to process asynchronous events, respond to requests, pre-load data, etc. C= provides all necessary means for that inside the same *parallel* operator: whenever a task tries to obtain its system-unique identifier (to be shared with other threads and to be used for thread interaction), the C= runtime automatically 'promotes' the task to an independent thread, and adds a new thread to the thread pool to continue fetching and executing other parallel tasks.

C= also provides special functions to asynchronously wake up such an independent thread and to wait for a wake-up signal, thus defining an easy-to-follow parallel thread interaction paradigm.

C= is unique in expanding the semantics of the *break* operator to let the primary task jump over the implicit barrier of a *parallel* operator and continue execution beyond, while other threads still execute the body of the *parallel* operator. That functionality is a key to instantly enabling asynchronous parallel execution in critical procedures that must return: e.g., C++ class constructors whose class implies asynchronous event handling.

3) Co-loading (heterogeneous parallelism)

C= also introduces a *coload* sub-operator which can be used only inside a *parallel* operator to indicate that enclosed instructions are to be executed by processors and co-processors simultaneously. The co-processors in this case are represented by C= runtime threads, which 'steal' (by indicating that they executed so many tasks and copying data the tasks depend on to the co-processor memory) work from the *parallel* operator's execution state and schedule kernels (formed from the instructions enclosed by the *coload* operator) for execution on the co-processors.

The C= visibility scope rules apply to *coload* regions, as well: variables inside the *parallel* scope but outside the *coload* scope are mapped to a co-processor's local memory shared between work items. That enables GPU-specific optimizations without introducing extra syntax elements.

Our current implementation relies on CUDA and OpenCL compilers to pre-compile *coload* operator bodies into kernels.

In the future, we plan to research the possibility of extending the *coload* operator to distributed environments: because a remote computer is in no way different from a co-processor (which is typically an independent computer plugged into a PCI slot), and can be similarly represented by a runtime thread that 'steals' work from *parallel* operators concurrently with local processors and co-processors.

B. Serial Operator

The *serial* operator encloses instructions to be executed atomically by multiple parallel tasks or threads. It is implemented as follows: a pointer to memory (on which multiple threads contend) is passed to a hash-function, which returns an index in a queue, on which contending threads arbitrate and where a loser thread stores its identifier, and then suspends itself. The winner thread, upon completion of the serial operator, fetches the identifier from the queue and resumes execution of the corresponding loser thread. As a result, the serial operator enables finest-granularity synchronization without introducing synchronization objects, and the probability of false conflicts between contending threads is controlled by the hash-function and the depth of the thread identifier queue.

Note that this scheme allows for serializing execution of any task or parallel region or thread (including threads created independently, not by C= operators). Also note that multiple nesting levels of serial operators are allowed.

IV. Performance Scalability

For performance testing we chose N-body simulation (described in [15]) as a compute-intensive benchmark that suits both symmetric and heterogeneous computer systems. Table I summarizes performance results for processing 16384 bodies.

TABLE I. C= Performance Data

System	Acceleration, times		
	1 CPU	***All CPUs***	***GPU***
Intel Core i7 Extreme 965 (Nehalem) @3.33GHz (**4 cores** with HT) + NVIDIA GeForce 9800 GX2	1	4.26	193.62
Intel Core i7 3667U (Ivy Bridge) @2.89GHz (**2 cores** with HT) + Intel HD Graphics 4000	1	1.85	52

We employed an optimization technique similar to one described in [15], which is based on computing interactions within smaller blocks of bodies and using local memory for storing intermediate results of in-block computations.

C= implementation of N-body simulation demonstrated good scalability (0.925 and 1.065 of theoretically expected acceleration at a given number of cores) on different CPU and GPU architectures, and maintained high precision of the output results: the square mean error (computed as a sum of per-element squared differences divided by the number of elements) was only 0.0000011, while some of the standard GPU samples written in other languages failed to produce consistent output results for the same input data and introduced square mean errors as high as 231.927045. Note that the results differed when samples were executed by CPUs and by 1 or 2 GPUs – detailed comparison is beyond the scope of this document.

V. Conclusion

We approached the problem of defining a minimal set of semantic categories that would allow for expressing the majority of parallel programming paradigms in imperative programming languages.

Now we consider some aspects of that problem solved: for instance, by introducing a single *parallel* operator that implements a flexible task-thread model and supports nested parallelism; and a *serial* operator that implements object-free synchronization. Other aspects are also considered solved and yet requiring syntactic polishing and/or automation: for instance, there may be no need for an explicit *coload* sub-operator any longer after C= gets natively supported by a compiler. While some aspects are considered a direction worth further research: for example, supporting distributed parallelism via the same *coload* operator as heterogeneous parallelism and transparently 'stealing' work from remote computers; allowing nested parallel operators inside a *coload* region; etc.

Currently, C=, as reported in detail in [1] and [2], is implemented as a multi-platform runtime library and syntax converters, and is used externally in [14]. Our future plans include implementing a C= compiler and extending our solution over to new operating systems and hardware architectures.

REFERENCES

[1] C= Specification and Reference Manual. Available online at http://www.hoopoesnest.com/cstripes/C=SpecMan.pdf

[2] C= binary package with programming examples. Available online at http://www.hoopoesnest.com/cstripes/C=.zip

[3] A list of parallel programming languages. Available online at http://en.wikipedia.org/wiki/List_of_concurrent_and_parallel_programming_languages

[4] Intel Threading Building Blocks (TBB). Available online at http://threadingbuildingblocks.org/

[5] Microsoft Parallel Patterns Library (PPL). Available online at http://msdn.microsoft.com/en-us/library/dd492418.aspx

[6] Intel Cilk Plus. Available online at http://software.intel.com/en-us/intel-cilk-plus

[7] OpenMP Standard. Available online at http://en.wikipedia.org/wiki/OpenMP and http://openmp.org/wp/openmp-specifications/

[8] OpenACC Specification. Available online at http://en.wikipedia.org/wiki/OpenACC and http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

[9] Microsoft C++ AMP (C++ Accelerated Massive Parallelism). Available online at http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx

[10] NVIDIA CUDA Documentation. Available online at http://docs.nvidia.com/cuda/index.html

[11] OpenCL Standard. Available online at http://en.wikipedia.org/wiki/OpenCL and http://www.khronos.org/opencl/

[12] POSIX Threads. Available online at http://en.wikipedia.org/wiki/POSIX_Threads

[13] Win32 Process and Thread API. Available online at http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847.aspx

[14] OpenCV Library. Available online at http://opencv.org/ and http://code.opencv.org/projects/opencv/wiki/ChangeLog

[15] L. Nyland, M. Harris, J. Prins. "Fast N-Body Simulation with CUDA". Available online at http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf

[16] M. Frigo, C. Leserson, K. Randall. "The Implementation of the Cilk-5 Multithreaded Language". PLDI '98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. ISBN:0-89791-987-4

[17] J. Reinders. "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism". O'Reilly Media, June 2007. ISBN 978-0-596-51480-8

[18] S. L. Olivier, J. F. Prins. "Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs". Available online at http://www.cs.unc.edu/~prins/RecentPubs/ijpp10.pdf

[19] A. Kukanov, M. Voss. "The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks". Published November 15, 2007. Intel Technology Journal. Volume 11, issue 04. ISSN 1535-864X DOI 10.1535/itj.1104.05. Available online at http://www.intel.com/technology/itj/2007/v11i4/5-foundations/4-tbb.htm

Natural C/C++ Parallelism

```
void salute()
{
    parallel()
    {
        int idx = pix();
        serial()
        {
            parallel(3)
            {
                printf("Hello, world, from task %d-%d\n",
                       idx, pix());
            }
        }
    }
}
```

A **single operator** to control **multiple parallel** programming **paradigms**

Natural C/C++ semantics and **variable visibility rules** and scopes

A single operator to control parallel synchronization

Clear means of parallel identification and interaction

All Parallel Paradigms in 2 Operators

Elegant Multitasking

```
std::vector<Data> data;
parallel(5000000)
{
  int i = pix();
  serial(&data[i])
  {
    data[i].process();
  }
}
```

Synchronized access to any data element without introducing synchronization objects

```
Stack
Single Execution State
{
  Task No. = 5000000;
  Code pointer;
  Registers;
}
```

Each thread from a pool decrements the task counter and "creates" a job to execute from a single execution state:

- No CPU oversubscription
- Dynamic work balancing
- Minimal memory footprint
- No task queue management overhead

Language-Friendly Multithreading

```
class X
{
  void* volatile id;
  X()
  {
    parallel(2)
    {
      void* pid = pid();
      if(pix())
      {
        id = pid;
        while(id)
        {
          wait();
          getMoreData();
        }
      }
      break;
    }
  }
};
```

A single operator to control multi-threading and multitasking

A real independent thread in a class constructor!

Getting a global ID promotes a task to **an independent thread**

Thread-0 returns, thread-1 waits until woken up by another thread/task

Reaching the break demotes a thread to a task

```
void X::read()
{
  wake(id);
  processData();
}
```

One Program Fits All

C= programs are executed concurrently by CPUs and GPUs

CPU CPU CPU

GPU GPU GPU

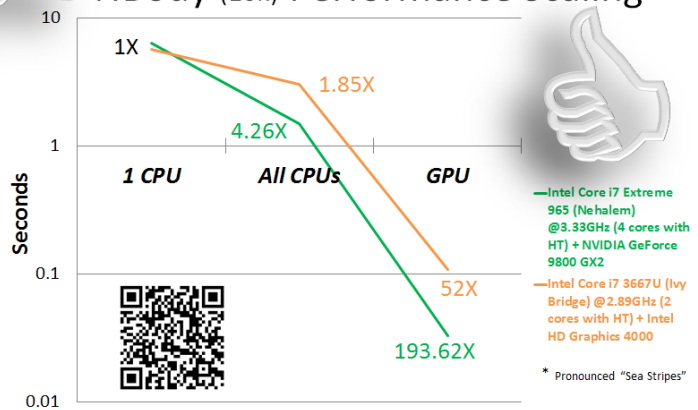
```
Memory
std::vector<Data> data;

parallel(data.size)
{
  coload()
  {
    data[pix()].process();
  }
}
```

```
Single Execution State
{
  Task No. = data.size;
  Code pointer;
  Registers;
}
```

Remote agents may concurrently "steal" the work from C= execution states and utilize their CPUs and GPUs

NBody (16k) Performance Scaling



C= is a step towards a **Unified Semantic Concept of Parallelism** to enable distributed heterogeneous programming with a single parallel operator

* Pronounced "Sea Stripes"