Computing Cloud or Cloud Computing?

Stanislav Bratanov
Stanislav.Bratanov@hotmail.com

***Abstract*—This paper introduces a new model of distributed parallel computations, referred to in the title as Computing Cloud. The model has a distinctive difference of being programmer-oriented, and having a special programming language (C= parallel C/C++ extension) as its basis. That brings in certain advantages, and enables creation of ad hoc computing networks, with dynamic work balancing. Such computing networks are inherently error-resilient and hardware architecture agnostic, which ensures high computational resource utilization (for instance, utilizing not only central processors, but graphics and other accelerators, as well), and dynamic computation topology – that is, the compute nodes of the network may be arbitrarily switched on and off. Besides, the nature of work distribution provides for commercialization and establishing an equitable billing system, when anyone who performed the computation gets paid exactly for the amount of data elements processed. All in all, we believe that the proposed model may be extremely useful in the emerging area of Internet of Things, as it enables software developers with unified means of organizing computations, and ensures said computations can be performed with equal ease both within datacenters and at home, utilizing free resources of in-home devices and appliances.**

I. Motivation

Internet of Things, among other things, is going to increase the demand for data processing and compute power, and that may impose a heavy burden on various miniature sensor devices. And that is one of the motivations behind thinking of a lightweight distributed computation environment, a sort of an ad hoc compute network, where more powerful devices can lend their resources to less powerful. Such a network should be dynamically self-balanced, error-resilient, and may in some cases have to be self-sufficient, not connected to any cloud services.

II. Requirements

Any solution in this emerging area, in order to get adoption, should be easy to use and maintain, be as unified as possible, that is, equally applicable to different computer architectures and, in particular, suitable for CPUs and GP-GPUs, have low development cost to programmers, be available free of charge (and probably open-sourced), and at the same time have a potential for commercialization.

III. Solution

The solution we propose is based on C= programming language (http://www.hoopoesnest.com/cstripes/cstripes-details.htm), which is currently used in OpenCV computer vision library and introduces an extremely simple programming model comprised of two primary operators: *parallel* operator to designate a scope of parallel execution, and *serial* operator to establish a synchronization point to access shared data. Some properties of said *parallel* operator make C= language look compelling for the purposes of our computing cloud proposal. Let's refer to the example of Fig. 1. As shown in the figure, *parallel* operator defines a scope of code to execute concurrently. An optional argument defines how many times the body of the *parallel* operator needs to be executed.

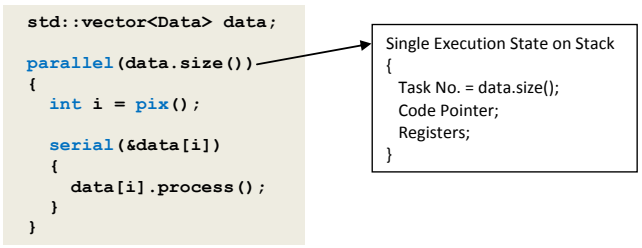

Fig. 1. C= programming paradigm defines just two primary operators – for parallelilization and serialization of work.

But unlike other parallel programming systems and languages, specifying that, for instance, the code is to be executed five or five million times does not lead to creation of as many tasks or threads. Instead, a special descriptor gets allocated on the stack to indicate which code and how many times has to be executed. That allows worker threads to execute the specified code concurrently by fetching the execution state (code pointer and registers) from the descriptor and decreasing the execution count. Thus, each worker thread assumes a new state and executes the body of the *parallel* operator. Note that the thread doesn't have to know anything about data the code is going to operate on, because the code itself detects all data locations and dependencies using a parallel identifier, unique to each invocation of the code.

All the above leads to another important property of C=: since each *parallel* operator simply creates an execution state and supplies an execution count, any other agent within the system can fetch that state for execution. And that effectively enables having, in addition to per-CPU worker thread pool, a dedicated thread per GP-GPU and other accelerators. Those threads may effectively "steal" work from parallel operators, by translating the code into a form a particular accelerator "understands", and do that concurrently with per-CPU threads.

For that purpose a special sub-operator – *coload* – was introduced (see the example of Fig. 2). It is also needed to provide hints on data dependencies, because those may not always be trivial to detect automatically.

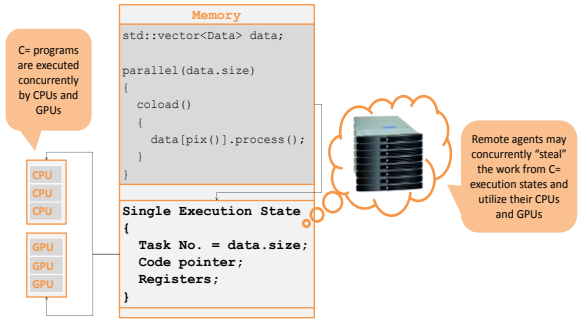

Fig. 2. Parallel regions can be executed by multiple local and remote agents concurrently (as indicated by the coload sub-operator).

The next important property of C= *parallel* and *coload* operators is the ability of remote execution. There may be a special agent running on a system that operates on behalf of another computer and concurrently "steals" work form *coload* operators and ships it over network to remote computers, where it then gets "stolen" by local CPUs and GPUs and/or by similar agents operating on behalf of other computers, and so on and so forth.

All that is exactly what enables the introduction of a computing cloud concept (see Fig. 3). Since C= utilizes a very limited amount of system resources (it's a natively compiled programming language), even the tiniest device will be able to make use of it for off-loading parallel computation to remote systems. More capable devices may run the C= co-load agent to share their compute power with those in need. Thus, your laptop or tablet may help processing data coming from a video surveillance system while you're browsing the web, and your fridge may readily assist your camera when its battery is low.

That already establishes an in-home computing cloud. Add a web service to register incoming and outgoing data processing requests (work "stolen" from *coload* operators), plus an optional storage and data transfer capabilities, plus a billing and accounting system – and we get a scalable solution for datacenters, companies, and individuals.

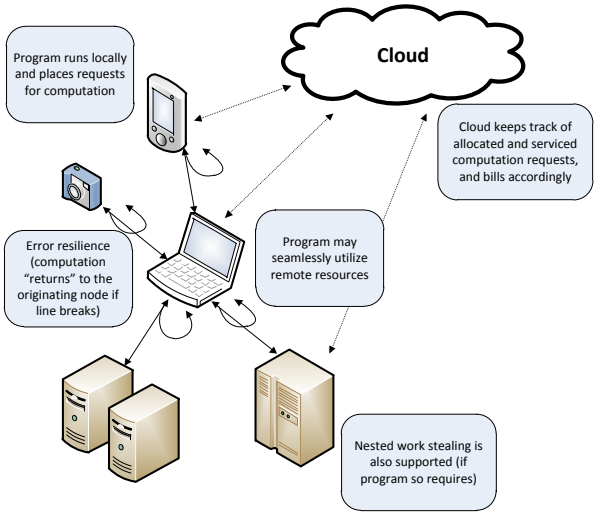

Fig. 3. Computing cloud as an arbitrary set of computation-resource providers, and a data exchange, coordinaiton, and billing service.

IV. Experiments

To prove the concept of an ad hoc computing cloud, we performed the following experiment: implemented N-body in the gravity field simulation task in C=, and set up two systems: one is an older laptop, not equipped with GP-GPU, and the other boasting the latest (at the time of experiment) CPU and GP-GPU, which made the latter system hundreds of times more powerful than the former.

Then, the N-body simulation was run on the first system, with millions of bodies as input parameters (to ensure that C= has to perform work balancing as that amount of bodies won't fit GPU memory) – and took several days to execute.

Then, the same experiment was run on the first system, but at the same time the second system was executing C= remote agent to establish remote work stealing – and the program finished in 30 minutes, because all the work was effectively executed by the second system's powerful GPU.

And finally, the same remote execution experiment was set up, but this time we were deliberately deleting all intermediate files generated by the C= runtime, switching on and off network connections, and disrupting the communication between the two systems by all possible means. As a result, the execution took a few minutes longer.

But after comparing the output of all the three experiments, we found the results to be the same (given the GPU floating-point rounding error), which proves the feasibility of the concept of a dynamically work balancing and error-resilient ad hoc computation network.

V. Advantages

The suggested computation service may have the following advantages: (1) lower entry barrier into the area of cloud computations and parallel programming by introducing unified means of parallel processing and work distribution, most naturally connected with popular C-like programming language syntax; (2) automatic work balancing between local and remote computing nodes; (3) improved error resilience, and automatic restart of computations; (4) secure computations as all data may be encrypted transparently to the communicating agents; (5) more equitable billing system based on the actual computations performed (the number of elements processed), not on time, number of processors, accelerators, memory, etc.

VI. Conclusion

We approached the problem of defining a common computation model for an arbitrary set of computing devices, and, as we believe, managed to prove the feasibility of our concept.

This research is still in progress, and so the goal of this poster is to inspire interest and discussion of new aspects of parallel distributed computations in light of Internet of Things that promises to generate tons of data which the communicating devices may not be able to process locally.

Another goal is to motivate further development of C= programming system, and find enthusiasts willing to push that project forward.

Computing Cloud or Cloud Computing?

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

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

Programming Time-Saver

```
void multiload()
{
    /// error-resilient programming facility
    int error = 0;

    /// split the input among CPUs and then run co-load from all of them
    parallel()
    {
        /// split the input
        int start = CPX_RANGE_START(N);
        int len = CPX_RANGE_END(N) - start;

        /// access through pointers (limitation of current coload syntax)
        float* shared xp = &x[start];
        float* shared yp = &y[start];
        float* shared zp = &z[start];

        /// process each part of the input data in parallel by all CPUs
        parallel()
        {
            /// execute the body of coload operator concurrently by
            /// CPUs and GPUs of local and remote systems,
            /// dynamically arbitrating for the data
            coload(
                   /// specify input and output data pointers and sizes
                   /// (will go away with the compiler support)
                   cpx_in   (float*, xp, len * sizeof(float)),
                   cpx_in   (float*, yp, len * sizeof(float)),
                   cpx_inout(float*, zp, len * sizeof(float)),
                   /// specify the data range to which the operator is applied
                   /// (will go away with the compiler support)
                   cpx_range(len),
                   /// specify a pointer to report an error or success
                   /// (may be reported as a return value of the operator
                   ///  if coload is natively supported by compilers)
                   cpx_error(&error))
            {
                int i, end;

                /// use standard macros for data decomposition
                /// (see examples of optimization in the accompanying samples)
                /// CPUs execute the entire loop below,
                /// GPUs execute loop iterations in parallel
                for(i = CPX_RANGE_START(len), end = CPX_RANGE_END(len);
                    i < end; i++)
                {
                    zp[i] += xp[i] + yp[i];
                }
            }
        }
    }
}
```

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

ADVANTAGES:

1) Lower entry barrier into the area of cloud computations and parallel programming by introducing unified means of parallel processing and work distribution, most naturally connected with popular C-like programming language syntax;
2) Automatic work balancing between local and remote computing nodes;
3) Improved error resilience, and automatic restart of computations;
4) Secure computations as all data may be encrypted transparently to the communicating agents;
5) More equitable billing system based on the actual computations performed (the number of elements processed), not on time, number of processors, accelerators, memory, etc.