Experiences Porting Mstack to ParalleX

Mark Pellegrini
Computer Architecture and Parallel Systems Laboratory
University of Delaware Department of Electrical and Computer Engineering
Newark, DE 19716, U.S.A
mpellegr@capsl.udel.edu
August, 2008

1. Abstract

Exascale computers are now a long-range goal in the high performance computing industry. Exascale computing will require 1000 to 10,000 times more parallelism than the current state of the art. To meet that need, Thomas Sterling's group at Louisiana State University are developing ParalleX, an architecture agnostic next-generation execution model.

I spent the summer of 2008 porting Mstack to ParalleX, the first application to be so ported. Mstack is a lightweight benchmark for evaluating parallel architectures, based median stacking, a signal processing technique. This paper documents my experience porting Mstack to ParalleX.

2. Introduction

With petaflop scale computers now a reality, exascale computers are now a long-range goal in the high performance computing industry. Exascale computing will require 1000 to 10,000 times more parallelism than the current state of the art. To meet that need, Thomas Sterling's group at Louisiana State University are developing ParalleX, an architecture agnostic next-generation execution model.

I spent the summer of 2008 porting Mstack to ParalleX, the first application to be so ported. Mstack is a lightweight benchmark for evaluating parallel architectures. Mstack is based median stacking, a signal processing technique. This paper documents my experience porting Mstack to ParalleX.

Section 3 describes the Mstack benchmark; section 4 details the motivation for, development status, and anatomy of ParalleX; section 5 describes my efforts to port Mstack to ParalleX; section 6 includes closing thoughts on the experience.

3. Mstack

Mstack is a small, easy-to-compile benchmark based on median stacking of data. Median stacking is a technique used in signal processing to filter noisy data. Median stacking has applications in photo astronomy [15] [12] and in the oil industry in processing reflection seismology data [8]. For the purposes of our benchmark, we choose to focus on the latter application. The Mstack benchmark was designed under the "keep it simple stupid" philosophy. It was designed to be practical to run on CPUs as well as on a wide variety of co-processors.

The real-world application that Mstack attempts to mimic is one stage of post-processing done on oceanographic seismic data in the oil industry. This data is collected by a ship, which drags a line of hydrophones behind it. As it moves, the ship puts out shockwaves (using an underwater sound generator – a metal box that is filled with gas which is then detonated). These waves of compressed and rarefied water bounce off the ocean floor ("subsurface") and reflect back upwards towards the hydrophones. The hydrophones measure the amplitude of the received waves. Using this data, geophysicists can reconstruct the subsurface topology.

With each explosion, each hydrophone measures reflections from a point on the subsurface halfway between the sound generator and the hydrophone itself. As the ship moves, explosions are set off at time intervals such that these subsurface points overlap.[1] With multiple datapoints corresponding to each sampled location on the subsurface, analysts have the ability to improve the signal to noise ratio by using a procedure called stacking. Ordinarily, stacking

1 In real uses of this technique, the hydrophones would sample many values per explosion. However, for the purposes of this benchmark, we are assuming they measure only one. Also note that for some set of values sampled during a single explosion by a single hydrophone, there may in fact be multiple spikes in the intensity of the reflected waves. These spikes correspond to disjoints in the density of subsurface layers.

involves simply averaging the measured values for a given point and depth; however, when there is noisy data, experience shows that the median stack provides a more accurate view of the subsurface topology. The measured values are stored in a three dimensional array, where the dimensions (m, l, and k) represent the ship's sailing vector, explosion number, and hydrophone number, respectively.

Mstack uses a bubblesort for its sorting algorithm. Bubblesort was chosen instead of a selection algorithm because (a) literature on the subject specifies sorting and not selection [11] [8] and (b) because for a simple selection algorithm, bubblesort will have the same computational complexity.[2]

A minimal amount of output is produced. This is helpful for debugging purposes and to disable overly aggressive compiler optimizations. In order to maximize the similarity between the Fortran and C code, the decision was made not to use pointers in either version. This means that no dynamic memory allocation was used in any of the versions and that array syntax is used throughout the C versions of the benchmark.

The Mstack reference version can be represented (in abbreviated form) in pseudocode by the following:

```
1  prompt_user(numchn, "How many channels do you want to sort (2–128)?")
2  for i from 1 to 2
3    for j from 1 to 3
4      initialize(traces, j)
5      //traces is a floating point array, 1001x1001x130
6      //j determines the dataset
7      //if j = 1, all elements in traces are initialized all 1.0
8      //if j = 2, all lists in traces are initialized in ascending order
9      //if j = 3, all lists in traces are initialized in descending order
10     for m from 1 to 1000
11       for l from 1 to 1001–numchn
12         for k from 1 to numchn
13           scratch[k] = traces [m][l–1+k][k];
14         bubblesort(scratch, numchn)
15         //sort the first numchn elements in scratch
16         traces[m][l][129] = median(scratch, numchn)
17         //return the middle value in the first numchn elements of scratch
18         //if numchn is even, return the average of the middle two
19     print traces[1][1][129]
20     print [1][800][129]
21     print traces[1000][1][129]
```

The entire program takes place inside the i and j loops (lines 2 and 3), which cause the program to iterate 2 times each over 3 sets of data. The data itself, which represents the values recorded by the boat's hydrophones, is written to traces, a three dimensional array (line 4). Values are then copied from diagonals in the traces array (in the L-K plane) into scratch.

Mstack is described in detail in [10] and [9].

4. ParalleX

ParalleX is an architecture agnostic next-generation execution model devised by Dr. Thomas Sterling and implemented by the ParalleX group at the Louisiana State University Center for Computation and Technology.

ParalleX's goals are (1) to expose massive program parallelism, on the scale sufficient to allow exascale computing, and (2) to provide automated system-wide latency hiding. To support this, the capstone of ParalleX will be Agincourt, a custom high-level high-productivity domain-specific parallel programming language. Agincourt's target domain is directed graphs. Although ParalleX's only officially supported high level language will be Agincourt, other established parallel programming languages (C/C++) may eventually have legacy support.

4.1. Background

ParalleX's draws on concepts from a number of earlier parallel computing paradigms: the actor model [6], Dataflow [14], the Threaded Abstract Machine (TAM) [3], the J Machine [7], and the HTMT project. [5]

ParalleX is not the first execution model to expose massive program parallelism and automatically hide latency. There were earlier execution models that did so, but they were unsuccessful and are no longer used. Why? Because high performance microprocessors in the form of MPPs and Beowulf clusters, combined with the first shared programming model, MPI, made it impossible for any other execution model to proliferate. In short, MPI is a "local maximum" that obstructs other, better parallel programming models.

In order to address Thomas Sterling's "four horsemen" of latency, overhead, resource contention, and resource starvation, current models of parallel computing use global barriers. These minimize the average overhead, but at the cost of limiting system scalability.

4.2. Anatomy of ParalleX

ParalleX accomplishes its goals (exposing parallelism and hiding latency, while doing away with global barriers so as to remain scalable) by message driven computation combined with a work-queue model.

4.2.1. Message driven computation In order for a computation on any computer system to proceed, the computation (that is to say, the program binary and program counter)

2 A simple general selection algorithm has $O(k \times n)$ runtime where k is the k-th largest or smallest element in the list (which we want to find). [1] If $k = \frac{N}{2}$ (the median element), the runtime will be $\frac{1}{2} \times N^2$, the same as bubblesort.

and data must be present at the same time at some given location. Traditionally, data is moved to the place of computation. ParalleX takes the opposite approach, preferring instead to move the computation to the location of the data, by means of parcels.[3] A parcel is akin to a remote procedure call, allowing one computation node to instantiate a thread on another. On arriving at that remote location, the parcel is converted into a thread. Conceptually, ParalleX treats the whole computer system like a processor-in-memory system. That is to say, send messages to the memory system telling it to do operations remotely on data that you (the programmer) don't want to copy locally.[4] This reduces contention on the memory system and spreads out the workload across the system, a technique known as split-phase transaction processing. This approach is particularly well suited to multicore and heterogeneous computing.

How does message driven computation differ from message passing (MPI)? Message passing consists of two or more remote computations influenced by partial results from each other.[5] In message driven computation, two or more remote computations pass messages telling the other how to operate on the data.

4.2.2. The Work-Queue Model ParalleX uses the work-queue model instead of the more traditional process model. In the process model, a process is instantiated on a processor, has its own address space, is static (starts at beginning of program, ends at end of program), uses IO space to exchange information, and uses global barriers for synchronization.

The work-queue model separates the work flow from the worker hardware. Worker hardware takes a sequence of dynamically allocated tasks. While executing these tasks, the worker may trigger remote tasks to be performed. The thread then dies, and is replaced by a new one from the queue. It may also go to sleep, although it is preferable to avoid this.[6] It is desirable to avoid going to sleep so that resources are not wasted by allocating them to a thread that is sleeping.

4.2.3. A Deeper Review - Elements of ParalleX At a deeper level, the key elements of ParalleX are:

- Parallel processes
- Threads
- Local control objects
- Distributed Global Address Space (DGAS)

Other factors are parcels, percolation, and copy semantics (copy semantics inspired by location consistency [4]).

ParalleX divides the system into a set of localities. A locality is defined as a location in the parallel computing hierarchy that has the ability to lock multiple resources atomically. Within a locality, actions have a bounded time - that is, there is a maximum amount of time that certions might take. Between localities, time is unbounded, meaning that there are no limits as to how long an action might take.

A process works across localities. A process consists of a set of threads for each locality, and their local control objects. In ParalleX, threads are lightweight - they are spawned often and die quickly, akin to fibers in the EARTH model. [13] Threads share a environment (an address space). A process can also contain sub-processes. Processes are hierarchical to prevent holes from occurring the global address space - if a sub-process exits, the parent thread inherits the sub-processes allocated resources.

```
Diagram of a ParalleX process
---------------------------------
| My threads:                   |
| T1 (on locality 1)            |
| T2 (on locality 1)            |
| T3 (on locality 2)            |
| T4 (on locality 2)            |
|                               |
| Subprocesses:                 |
| Subprocess #1 (T1, T2, T3...) |
| Subprocess #2 (T1, T2, T3...) |
|                               |
|  LCO #1 - mutex               |
|  LCO #2 - dataflow            |
---------------------------------
```

ParalleX has a distributed global address space (DGAS).[7]

```
Spectrum of approaches to parallel
memory systems:
Less coherent <------------------> More coherent
              |    |     |     |
             DSM  DGAS  PGAS   DM
```

- Distributed system memory (as used in clusters, MPPs) - Not cache coherent
- Distributed Global Address Space (as used in ParalleX) - Global address space addresses are dynamically mapped to local address spaces. This allows addresses to migrate. Not cache coherent.
- Parallel Global Address Space (as used in UPC) - Global address space addresses are statically mapped to local address spaces. Not cache coherent.

3 However, ParalleX does support the traditional data-to-computation model as well.

4 For example, "To memory system: Compute the eigenvalue of matrix X and return the result to me."

5 For example, a heat diffusion computation.

6 But the ability to sleep is semantically necessary.

7 See section 4 of [2] for a good overview of current approaches to parallel memory systems

- Distributed memory - A cache coherent global address space.

In ParalleX, global address lookups are provided by the DGAS service. The DGAS service is implemented in HPX, the lowest layer of the ParalleX toolchain. It takes a 128 bit global identifier (GID) and returns a locality and an LVA. A locality is the network address and port (known to network engineers as the 5-tuple) of the machine identified by the GID, and the LVA is the local address space value identified by the GID. In HPX, anything can be assigned a global identifier (GID) - RAM, threads, co-processors, local control objects, etc. Anything which can receive a parcel is a first class object.

Local control objects (LCOs) are the means by which synchronization is implemented. All thread blocking is done using local control objects. Local control objects provide the programmer with barriers, virtual full-empty bits, and other forms of synchronization objects. LCOs have object-oriented properties - they contain their own data and methods. One important observation Sterling's group has made is that a sleeping (aka, depleted) thread has the same semantics as an LCO - both are waiting on some external event to re-activate.

To invoke a procedure call in HPX, a thread uses the applier. The applier is a function that determines whether or not a function call should be invoked locally or not. To determine this, the applier queries the DGAS service. If the function needs to be invoked locally, the applier does so; if not, it sends a parcel to the correct remote location.

A parcel in HPX is like a remote procedure call, providing asynchronous communication. A parcel contains:

- A destination GID
- A source GID
- An action - which consists of: (1) A function to be invoked, and (2) arguments to that function
- A continuation - a list of GIDs of local control objects

The action consists of the function be invoked on the remote machine, and the arguments to that function. A continuation tells the remote location what to do after the procedure call finishes. If the procedure returned a value, that value is returned to those locations specified in the continuation; if it did not return a value, an empty parcel is sent to those locations (equivalent to the return from a void function).

On a traditional parallel architecture, it is difficult for the programmer to efficiently parallelize code if he lacks important knowledge of the target architecture. Bearing in mind that ParalleX is architecture agnostic, details of the target architecture are not known when compiling into HPX source code. At runtime, the compiler or programmer choose policies that determine how the HPX runtime system behaves in several important ways.

4.3. ParalleX Development Status

The ParalleX toolchain is as follows:

- High level programming language: Agincourt
- Intermediate representation: PXIF
- Low level runtime system: ParalleX runtime system – currently HPX, whose prototype was known as DistPX. The ParalleX architecture is specified so that others groups may create alternative ParalleX runtime systems.
- Lowest level: Compiled C++ code

As of this writing, Agincourt is in the formative stage of development - existing on paper and in its programmers' minds only. PXIF is in a more advanced stage of development - the specification is nearly finished (although still incomplete), with some source code written. During my time at LSU, neither were available for my use.

HPX is the ParalleX runtime system. HPX currently resides between the operating system and the application, although future versions of HPX may be part of the operating systems itself. The goal of HPX is to provide essential services to the above layers. An important feature of HPX is that it makes a variety of underlying architectures identical to the programmer, much as a virtual machine does.

HPX provides communication, remote procedure call routines, and global address functionality to the programmer. HPX is in the experimental stage of development - it has an existing code base that builds off of early work done on DistPX (HPX's predecessor), but still has serious developmental limitations - missing features, known bugs, et cetera.

Ideally, the toolchain can target any architecture, however, at this time the target architecture is the commodity cluster.

5. Porting Mstack to ParalleX

One current HPX limitation is that all parcels (remote procedure calls) must operate on class methods. So, the first step in the porting process was to rewrite the Mstack reference version into C++ using classes.[8]

The Mstack class had the following prototype:

8 The source code for all Mstack versions is on the web at `http://www.eecis.udel.edu/~mpellegr/mstack/`. The C++ class-based port of the reference implementation is available at `http://www.eecis.udel.edu/~mpellegr/mstack/mstack_classes.cpp`

```
1  class traces{
2    private:
3      float mytraces [1001][1001][130];
4      int numchn;
5      int sort_and_store(int, int);
6    public:
7      traces (int);
8      int initialize (int);
9      int do_run();
10 };
```

The second step in the porting process was to rewrite the Mstack source code to use the HPX API and Boost. Boost (`http://www.boost.org/`) is a set of open source libraries for C++ that provide much functionality beyond that provided by the language itself. HPX is closely coupled with boost.

During this process, I worked closely with Hartmut Kaiser, the main HPX developer. During the transition, the class was renamed from traces (shown in the source code above) to mstack.

One that was finished, the program flow was as follows:

- main() - runs one time on the master node. It takes command line arguments and uses those arguments to set up the DGAS server, then main calls hpx_main.
- hpx_main() - prompts the user for the number of channels for mstack. hpx_main then forks 6 threads on the current (master) node. Each of those threads runs the onepass function
- onepass() - onepass is run 6 times concurrently on the master node. Each instance of onepass remotely invokes the do_run function of the mstack class. Each onepass instance then goes to sleep, until the do_run function returns. On waking, each instance prints the three values returned by the do_run function (encapsulated in the return structure, of type ret_val).
- mstack::do_run (j) - do_run takes as an argument integer j, whose value is from 1 to 6 inclusive. The j value corresponds to Mstack's outer i/j loop. (See source code in section 3). do_run is instantiates 6 times concurrently on remote nodes (that is to say, nodes other than the master node). do_run initializes the class, and performs one pass of the mstack benchmark - stacking all values in one iteration of the i/j loop and returning the three proscribes values.

The "factory" - the HPX component responsible for instantiating (allocating RAM, requesting a GID) a function on remote nodes - is currently unable to do so dynamically. Currently, in order for it to instantiate a function, that function must be hard-coded into HPX. This will be remedied in the near future, but as a temporary work-around Hartmut and myself hard-coded the do_run function into HPX. In spite of that, however, the program hangs during HPX API call to instantiate do_run. As of this writing, the cause of this problem is unknown but very likely lies in HPX.

6. Closing Thoughts

ParalleX has the potential to greatly advance the state of high performance computing, addressing known, unsolved problems in our current approaches to large supercomputers.

Performance tests (especially in regards to its scalability) are needed before one can draw conclusions to its viability as a real-world solution. Before that can happen, however, the toolchain will much more work to become mature.

While porting Mstack, I noticed the following issues with the Mstack toolchain that I believe need to be addressed:

- Blocker-level issue: Mstack hangs at runtime when using HPX's eager_future function to remotely fork instances of mstack::do_run. The reason for this is as-yet unknown.
- Highly important issue: Several people mentioned that HPX has bug whereby C++ iostreams cause a segfault, for as yet unknown reasons.
- Highly important issue: The "Factory" lacks the ability to dynamically create components. (Described in detail in section 5)
- Medium issue: HPX lacks a function retrieve a GID for one computation node. HPX lacks a "give me 1 GID" function, which takes no arguments, and return the GID of a computation node where work can be sent to be processed. Ideally, it should return the computation node with the lowest load. However, because the function may be called multiple times before the first parcel is sent out, returning the lowest load CPU may cause too many parcels to be sent to one destination. (Therefore, it is probably best to return GIDs in round-robin fashion) Mstack currently implements a hack to avoid this problem (under the comment "get all non-local GIDS and store six of them", lines 165-181). It iterates through all GIDs, and stores the first few (6 in Mstack's case) for later use.
- Small issue: The DGAS runtime should have a command-line option to run a DGAS service only. The DGAS runtime, as it is currently written (in `https://svn.cct.lsu.edu/repos/projects/parallex/trunk/hpx/runtime/hpx_runtime.cpp`) always launches both a full HPX runtime and the DGAS service (on two different ports). The former of these is likely to lead to a misallocation of work. (It is roughly equivalent to running heavy duty scientific code on the department

mail server) The hpx runtime should be modified so that it has a command line option to launch only a DGAS server. I discovered this problem because when using the hack described in (2) above, I ended up doing exactly that - running Mstack source code on the DGAS server. At Hartmut's suggestion, I have created a pure (DGAS only) runtime. The source is at `https://svn.cct.lsu.edu/repos/projects/parallex/trunk/hpx/runtime/hpx_runtime_pure.cpp`

- Small issue: Cmake does not compile the HPX toolchain with built-in support for debugging. Unless there is negative impact I'm not aware of, all Cmake files in HPX should by-default compile binaries with built-in support for debugging.

7. Acknowledgements

The Mstack benchmark was created by Daniel Pressel, of the United States Army Research Laboratory. He created it based on experiences and knowlege he gained while previously employed in the oil industry, processing sampled data.

This work would not have been possible without the daily support of the ParalleX group at Louisaiana State University's Center for Computation and Technology - Dylan Stark, Chirag Dekate, Anshul Tandon (now at Google), Richard Guidry, Riley Andrews (now at Georgia Tech), Steven Brandt, Maciej Brodowicz, and Hartmut Kaiser. Hartmut worked closely with me to port Mstack to HPX; Dylan, Chirag, and Anshul were always available to answer questions of mine.

Special thanks go to Thomas Sterling, for giving me the opportunity to work with his group.

References

[1] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[2] S. Boggan and D. Pressel. GPUs: An Emerging Platform for General-Purpose Computation. In *ARL-SR-154*. U.S. Army Research Laboratory, 2007.

[3] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Tam—a compiler controlled threaded abstract machine. *J. Parallel Distrib. Comput.*, 18(3):347–370, 1993.

[4] G. R. Gao and V. Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical report, IEEE Trans. on Computers, 1998.

[5] G. R. Gao and K. B. Theobald. The htmt program execution model. Technical report, In Workshop on Multithreaded Execution, Architecture and Compilation (in conjunction with HPCA-4), Las Vegas, 1997.

[6] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[7] M. D. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: An architectural evaluation. In *In Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, 1993.

[8] O.E. Naess and L. Bruland. Improvement of multichannel seismic data through application of the median concept. *Geophysical Prospecting*, 37(3):225–241, 1989.

[9] M. Pellegrini. A Case Study of the Mstack Cross-Platform Benchmark on the Cray MTA-2. Master's thesis, University of Delaware, May 2008.

[10] M. Pellegrini and D. Pressel. Mstack: A Lightweight Cross-Platform Benchmark for Evaluating Co-processing Technologies. In *ARL-MR-0683*. U.S. Army Research Laboratory, 2007.

[11] D. Rawlings, R. Korsch, B. Goleby, G. Gibson, D. Johnstone, and M. Barlow. The 2002 Southern McArthur Basin Seismic Reflection Survery. Technical report, 2002.

[12] J. Schwartzenberg, S. Phillipps, and Q. Parker. Digital stacking of Tech Pan films and the photometry of faint galaxies. *Astronomy and Astrophysics Supplement Series*, 117:179–187, May 1996.

[13] K. Theobald. *EARTH: An Efficient Architecture For Running Threads.* PhD thesis, McGill University, May 1999.

[14] P. Whiting and R. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, Winter 1994.

[15] X. Zheng, E. Bell, H. Rix, C. Papovich, E. L. Floc'h, G. Rieke, and P. Perez-Gonzalez. Detecting Faint Galaxies by Stacking at 24 μm. *Astrophysical Journal*, 640:784–800, April 2006.