GSoC ‘18 Project Proposal

Organization: STE||AR Group

A C++ Runtime Replacement

PERSONAL DETAILS

Name and Contact Information:

- **Name:** Nikunj Gupta
- **Email:** nikunj.nkgpta@gmail.com
- **IRC Nickname:** nikunj
- **Phone Number:** +91 7503768269
- **Github:** NK-Nikunj
- **Skype:** nikunj.nkgpta

University and Current Enrollment:

- **University:** Indian Institute of Technology Roorkee
- **Degree Program:** Bachelor of Technology (B. Tech)
- **Field of Study:** Computer Science and Engineering (Batch of 2021)

Meeting with Mentors:

- Reachable anytime easily through email or IRC. A well planned video chat session if required.
- Available for video chat anytime between 10:00 am - 4:00 am IST during vacations and 6:00 pm - 4:00 am IST otherwise.

Background Information

Coding Skills:

I am very comfortable with all the technologies that are to be used in this project. I can write (in order of proficiency):

- C++
- C++ Standard Library, Git
- Boost C++ Libraries
- CUDA (no prior experience but willing to learn as required)

I will be using Doxygen as my documentation tool.

Development Environment:

- Ubuntu 16.04 LTS with HPX Built and Installed
- Highly proficient in using vim, atom

System Configuration:

- Currently using HP Omen 15t with following major configurations:
    - Intel Core i7 7700HQ
    - Nvdia GTX 1050ti
    - 16GB DDR4 2400MHz RAM
    - 128GB + 1TB SSD + HDD Combo

About Me:

I am a freshman currently enrolled in Computer Science and Engineering at IIT Roorkee. I have developed a passion for programing and web development during my high school year and from then, most of my time goes into reading and writing software. I am relatively new to Open Source World.

I have the experience of working closely with a team as I am an active member of Information Management Group at IIT Roorkee, a group of enthusiasts who manage the institute main website, the Internet and intranet activities of the university.

I have worked on a few projects one of which is Teesa, a Decentralised Payment Infrastructure especially aimed towards P2P Government Payment settlements which keeps each transaction transparent, immutable and irreversible. It is a fully decentralised app with smart contracts. Most of my C++ development and projects have been in the form of assignments (during my high school, currently not available online). I have experience in implementing various data structures in C++ and feel absolutely comfortable while writing code in C++.

My recent interest in parallel and distributed programming led me to HPX. The HPX model intrigued me and therefore to understand it better I dived into the source code. The project I'm interested in is in sync with my interest in runtime systems and their implementation. Through implementation of this project, I will gain a deep insight of the working of a runtime system. More importantly I will gain incredible knowledge in runtime initialization sequencing techniques and implementation.

I have read various research papers, documents and watched videos to better understand the HPX runtime system and gain a better understanding of HPX in general. Contributing to HPX will enable me to better understand the library structure. Post GSoC, I will make sure that my code is thread safe. Also, I will work on implementing various examples to check for robustness of the runtime system. Through contributing to such a large code base, I will gain experience in how big libraries are developed.

Pre-GSoC Involvements:

Here is a list of PRs I've worked on and Issues I've raised:

- #3178 (merged): Fixed a typo in Docs and Implemented the build of hello_world_component in Circleci
- #3231 (mergeable): Implemented a prototype for my GSoC project.
- #3240 (merged): Fixed a few typos
- #3208 (issue): Raised issue regarding init_globally always throwing a runtime error (fixed after the issue was raised)
- #3226 (issue): Raised issue to explain that memory_block was breaking, resulting in runtime error for quicksort. Currently working on reimplementing memory_block and parallel quicksort example.

Project Proposal

Introduction:

Currently the HPX runtime system needs to manually "lift" regular functions to HPX threads in order to have all the information for user-level threading available.

This is implemented in 3 broad ways:

- Using `hpx_main` header which uses a macro to initialize the runtime system.
- Using `hpx::init()` which starts the HPX runtime environment and suspends the main os-thread.
- Using `hpx::start()`, `hpx::suspend()`, `hpx::stop()` model.

Both second and third way of implementing the HPX runtime includes calling the functions from main. These methods rely on providing HPX entry point as `hpx_main` function or any other user defined custom function. Since these methods are invoked from C-main, very few HPX functionalities can be invoked from C-main.

While using `hpx_main` header is a very elegant way of initializing HPX runtime with seamless integration, it relies on a macro `#define main hpx::user_main()` which can produce unexpected behavior.

My project is to investigate and implement another runtime system which is as elegant as the `hpx_main` initialization with robustness matching that of `hpx::init()` and `hpx::start()`.

Proposed Solution:

After a considerable amount of research, I found 3 major ways to implement the runtime system. These are:

- **Designing custom hooks:**
    - This would involve creating hooks to dynamically link the HPX runtime system with the current program.
    - While this is an elegant solution, this would involve implementing different custom hooks for different environments (i.e. it depends on the Operating System).
    - Thus, this will involve implementing same hooks for different OS. This would involve duplication of code, which I feel is not a good practice.

- **Overriding the Compiler initialization function stack (Can be a viable option):**
    - As the heading suggests, this method would include overriding the initial compiler entry functions such as `_start`, `__libc_csu_init`, `__libc_start_main` and re-implementing them such that it initializes the HPX runtime system as well.
    - This would involve linking with -nostartfiles and then providing custom entry points in our own linker script to initialize main as an HPX thread.
    - This would not be an elegant way of implementing HPX runtime system as a lot of pre-implemented code will have to be re-implemented. This would therefore involve writing redundant code, which again is not a good practice.
    - A further investigation is necessary during homework period.

- **Using Global initialization (Preferred way):**
    - This would involve creating a struct (definition given in later is sections) which initializes the HPX runtime system before program enters the main function. This would directly register the C-main as an hpx thread.
    - This is a very elegant way to initialize the runtime system. However, it has a few shortcomings which needs to be tackled. These are discussed in detailed in the later sections.

Proposed Implementation:

The complete implementation can be summarized in 2 ways:

- Creating a global struct with required implementation (struct diagram given in detail below) to generate the HPX runtime system. This struct with then be called with a global object.
- Since we are initializing the HPX runtime globally, it would involve initialization sequence issues which needs to be resolved.

Detailed Explanation:

- The first implementation is to initialize the command line arguments such that `hpx::init()` can be called with it. Structure for the same can be implemented as follows: (Has been directly taken from init_globally example in quickstart)

```
#if defined(linux) || defined(__linux) || defined(__linux__)

int __argc = 0;
char** __argv = nullptr;

// Function to set command line arguments in linux based systems
void set_argc_argv(int argc, char* argv[], char* env[])
{
    __argc = argc;
    __argv = argv;
}

// Attaching a function pointer to above function in preinit_array section
__attribute__((section(".preinit_array")))
    void (*set_global_argc_argv)(int, char*[], char*[]) = &set_argc_argv;

#elif defined(__APPLE__)
```

```
#include <crt_externs.h>

// Function to get the argc value
inline int get_arraylen(char** argv)
{
    int count = 0;
    if (nullptr != argv)
    {
        while(nullptr != argv[count])
            ++count;
    }
    return count;
}

int __argc = get_arraylen(*_NSGetArgv());
char** __argv = *_NSGetArgv();

#endif
```

- Second implementation is to call init from a global initializer. It can be implemented as follows:

```
struct __manage_global_runtime {
       __manage_global_runtime() {

       // Correct the initialization sequence before running hpx::init()
       // ...
       // Set configuration for runtime system
       std::vector<std::string> const cfg = {
            // allow for unknown command line options
            "hpx.commandline.allow_unknown!=1",
            // disable HPX' short options
            "hpx.commandline.aliasing!=0"
        };

        using hpx::util::placeholders::_1;
        using hpx::util::placeholders::_2;
        hpx::util::function_nonser<int(int, char**)> start_function =
            hpx::util::bind(&main, _1, _2);

        // Initialize the runtime system here
        hpx::init(start_function, __argc, __argv, cfg, hpx::runtime_mode_console);
       }

      // Here goes registration wrapper functions

       ~__manage_global_runtime() {
              // Destructor functions goes here, meant to de-initialize everything
        }
};

__manage_global_runtime __init_hpx;
```

- Create a destructor struct which shall call `std::exit()` to safely exit. Simple struct diagram is as follows:

```
struct __destruct {
        __destruct() {
               // Signal the runtime system to exit successfully
                std::exit(EXIT_SUCCESS);
        }
};

__destruct __destroy;
```

- The next step will be to investigate the exact runtime initialization sequence such that the code does not break. One of such initialization issues include one from function registrations on HPX threads (those created on os kernel threads). To implement this we can create struct wrapper. One simple implementation is of the form:

```
Struct registration_wrapper {
    // functions with registration type goes here.
    // For example, registering functions and threads.
    // All of them call functions inside of __manage_global_runtime with init object
};
```

Issues:

There are some issues in this implementation which needs rectification. These are as follows:

- The global object initialization runs on hpx::init(). This means the compiler thread is suspended until hpx::finalize() is called. This implies that any user created global object will never be initialized.
- Since HPX itself initializes global object of it’s own, there is no guarantee that those will get initialized properly. This can be rectified with further investigation which will involve to properly understand their initialization sequence and then maintaining the exact initialization sequencing order.

- Finally many of HPX components initialization sequencing order will be altered in some way or the other using global object initialization. This can be rectified by carefully investigating their initialization sequence and accordingly making changes to correct them.

Wishlist:

Can be implemented if all the tasks are completed successfully and the GSoC program is yet to end. It can also be taken up post GSoC.

- Register any kernel thread with HPX runtime system without explicitly calling register functions.
- Implement all examples from examples directory with implemented runtime system.

Proposed Timeline

I have my end semester examinations between 26th April - 5th May. I will be a bit inactive between 11th April to 6th May for exam preparations.

I believe I have enough fuel to get started on my goals - as a result to my involvement with the organization for almost two months now. So I will be setting grounds early to avoid stopgaps during the actual GSoC period. There are a couple of weeks before the actual timeline where I would make sure that I have done all preparatory work for the project. All dates mentioned here are weeks - starting from Monday and ending on Sunday. I'll be writing blogs every week, and I'm not including it in schedule (to prevent redundancy).

27 March - 17 April ("Homework" time):

- Investigate if the second option is a viable option and further if it can be implemented.
- Test the robustness of the prototype runtime system I created
- Investigate the exact initialization sequence
- Discuss the sequence with mentors

18 April - 6 May (Not-so-Active Period):

- Interact with the members of the community
- End semester exams will be near so, I'll be a bit inactive, though always available on email and IRC or any other means of communication. At times, I'll try solving or finding issues or play around with HPX source code.
- May 5 - 6, I'll be packing up from the University campus and moving home, so mostly inactive these two days.

7 May - 13 May (Community Bonding and “Homework” time):
- Continue investigating the initialization sequence issues
- Discussing the necessary time allotment with mentors

14 May - 20 May (Week 1):
- Start work with full enthusiasm.
- Create the file structure in accordance with above code
- Create the main initialization file as `hpx_initialize.hpp` and add it to include directory
- Build the system to check for any breakage

21 May - 27 May (Week 2):
- Document all the code written till now and add it to docs
- Implement few examples from examples directory with the proposed runtime system.

28 May - 3 June (Week 3):
- Work on correcting initialization sequence issues in registration functions found in `hpx::util`
- Implement few examples from examples directory with the proposed runtime system.

4 June - 10 June (Week 4):
- Complete the registration initialization issues.
- Document the code written till now and add it to docs.

11 June - 17 June (Week 5):
- **Phase 1 evaluation**
- Start working on initialization issues with HPX command line option initialization issues.
- Correct initialization issue for HPX command line options, starting with --hpx:help, --hpx:version, --hpx:info, --hpx:options-file

18 June - 24 June (Week 6):
- Produce working results for above initialization issues.
- Correct initialization issues for HPX configuration command line options

25 June - 1 July (Week 7):
- Produce working results for above initialization issues.
- Correct initialization issues for HPX performance counters command line options.

2 July - 8 July (Week 8):
- Produce working results for above initialization issues.

- Check for robustness of system with implementing few examples from examples directory with the created runtime system.

9 July - 15 July (Week 9):
- **Phase 2 evaluation**
- Start working on initialization issues within the HPX system.
- Correct initialization issues (if any) in `hpx::parallel`, `hpx::components`

16 July - 22 July (Week 10):
- Implement examples based on `hpx::components` and `hpx::parallel` to check for robustness of the system.
- Correct initialization issues (if any) in `hpx::lcos`, `hpx::compute`

23 July - 29 July (Week 11):
- Implement examples based on `hpx::lcos`, `hpx::compute` to check for robustness of the system
- Correct initialization issues (if any) in other HPX functions

30 July - 5 August (Week 12):
- **Tentative wrap up - cosmetic refinements**
- Update documentation where ever necessary

6 August - 12 August (Week 13):
- **End Term evaluation**
- Prepare content for end term evaluation, including deliverables from the wish list.

13 August - 21 August (Week 14):
- Buffer week and conclusion of GSoC

<u>Availability:</u>

My vacations start from 7 May and end on 15 July. The official GSoC period is from 14 May to 14 August. I can easily devote 60-70 hours a week till my college reopens and 40-50 hours per week after that. I'm free on weekends and mostly free on Wednesdays. I intend to complete most of the work before my college reopens.
Other than this project, I have no commitments/vacations planned for the summer. I shall keep my status posted to all the relevant community members on a weekly basis and maintain transparency in the project.

Post GSoC

I’m interested in working with STE||AR GROUP even after GSoC ends. I’ve been contributing to HPX for over a month now and I’ve really loved the experience. I got familiar with the community and I believe I’ve learnt a lot interacting with the mentors. I feel this kind of mentorship is necessary. I’ll be an active member in the community and keep contributing. My motivation would always be that I’d be able to contribute to any issue that comes up. This gives me a lot of satisfaction.

I will also work on my wishlist and keep my implemented runtime system issue free. I will work on issues other than the ones relating to my GSoC project.

****