مثال اول

مثال زیر از *SuperGlue* برای اجرای یک وظیفه[1] منفرد غیروابسته استفاده کرده است.

```
#include "sg/superglue.hpp"
#include <iostream>

struct Options : public DefaultOptions<Options> {};

struct MyTask : public Task<Options> {
    void run() {
        std::cout << "Hello world!" << std::endl;
    }
};

int main() {
    SuperGlue<Options> sg;
    sg.submit(new MyTask());
    return 0;
}
```

برای کامپایل کردن با استفاده از *g++* به صورت زیر می توان عمل کرد

```
g++ -I $(SUPERGLUE)/include helloworld.cpp -pthread
```

توجه: *$(SUPERGLUE)* باید با آدرس دایرکتوری ای که *SuperGlue* در آن قرار دارد جایگزین شود.

شرح مثال

ابتدا، فایل سرآیند اضافه شده است.

```
#include "superglue.hpp"
```

یکی از اهداف *SuperGlue* داشتن امکان سفارشی سازی است. این هدف با داشتن ساختار *Options* که پارامتر قالبی برای اغلب کلاس‌ها است، پیاده سازی گردیده است. در این مثال، از ساختار *Options* پیش‌فرض استفاده شده و به صورت زیر تعریف می گردد.

```
struct Options : public DefaultOptions<Options> {};
```

وظایف به طور معمول هر ساختار داده اشتراکی را که به آن دسترسی دارند، در سازنده کلاس ثبت می کنند. در این مثال وظیفه هیچ وابستگی ندارد، به همین دلیل به سازنده نیز نیاز ندارد. این وظیفه به شکل زیر تعریف شده است.

```
struct MyTask : public Task<Options> {
    void run() {
        std::cout << "Hello world!" << std::endl;
    }
};
```

سیستم زمان‌اجرا[2] و ریسمان های کارگر[3] با ایجاد کردن نمونه‌ای از شئ کلاس *SuperGlue* آغاز می‌شوند. در کد زیر این مساله نشان داده شده است.

```
SuperGlue<Options> sg;
```

کد بالا به ازای تمام هسته های پردازشی موجود در سیستم (جز آن هسته ای که برای ریسمان اصلی رزرو شده است) یک ریسمان کارگر ایجاد می کند.

همچنین می‌توان تعداد پردازنده‌های مورد استفاده را در سازنده کلاس محدود کرد. برای محدود کردن تعداد پردازنده به ۴ عدد(یک ریسمان اصلی به همراه۳ ریسمان کارگر) سیستم زمان‌اجرا به شکل زیر شروع می‌شود.
توجه: تعداد پردازنده‌های مشخص شده باید از تعداد هسته‌های پردازشی سیستم کمتر یا مساوی باشد.

1 . Task
2 . Run-time system
3 .Worker thread

```
SuperGlue<Options> sg(4);
```

وظایف با استفاده از متد *submit()* به *SuperGlue* ارسال می شوند.

```
sg.submit(new MyTask());
```

هر زمان که وظیفه ای به *SuperGlue* ارسال شود، مالکیت (مسئولیت حذف آن) به *SuperGlue* انتقال داده می‌شود.
زمانی که اشیا *SuperGlue* به پایان محدوده خود می‌رسند، بندهای[4] ضمنی وجود دارند که منتظر اتمام تمامی وظایف شده و به کار ریسمان‌های کارگر پایان می دهند.

وظایف با وابستگی‌ها

هدف اصلی *SuperGlue* مدیریت بین وابستگی‌های وظایف به‌شکلی منعطف و کارا است. مثال زیر نمونه‌ای از وظیفه با وابستگی است.

```
#include "sg/superglue.hpp"
#include <iostream>

const size_t numSlices = 5;
const size_t sliceSize = 100;

struct Options : public DefaultOptions<Options> {};

// Task that inputs a vector and outputs a scaled vector.
struct ScaleTask : public Task<Options> {
    double s, *a, *b;
    ScaleTask(double s_,
              double *a_, Handle<Options> &hA,
              double *b_, Handle<Options> &hB)
    : s(s_), a(a_), b(b_)
    {
        register_access(ReadWriteAdd::read, &hA);
        register_access(ReadWriteAdd::write, &hB);
    }
    void run() {
        for (size_t i = 0; i < sliceSize; ++i)
            b[i] = s*a[i];
    }
};

// Task that input two vectors and sums them into an ouput vector
struct SumTask : public Task<Options> {
    double *a, *b, *c;
    SumTask(double *a_, Handle<Options> &hA,
            double *b_, Handle<Options> &hB,
            double *c_, Handle<Options> &hC)
    : a(a_), b(b_), c(c_)
    {
        register_access(ReadWriteAdd::read, &hA);
        register_access(ReadWriteAdd::read, &hB);
        register_access(ReadWriteAdd::write, &hC);
    }
    void run() {
        for (size_t i = 0; i < sliceSize; ++i)
            c[i] = a[i]+b[i];
    }
};

int main() {
    double data[numSlices][sliceSize];

    for (size_t i = 0; i < sliceSize; ++i)
        data[0][i] = 1.0;

    // Define handles for the slices
    Handle<Options> h[numSlices];

    SuperGlue<Options> sg;
    sg.submit(new ScaleTask(2.0, data[0], h[0], data[1], h[1]));          // h_1 = 2*h_0
    sg.submit(new ScaleTask(3.0, data[0], h[0], data[2], h[2]));          // h_2 = 3*h_0
    sg.submit(new SumTask(data[0], h[0], data[1], h[1], data[3], h[3])); // h_3 = h_0+h_1
    sg.submit(new SumTask(data[1], h[1], data[2], h[2], data[4], h[4])); // h_4 = h_1+h_2

    // Wait for all tasks to finish
    sg.barrier();
```

4 .Barrier

```
    // The data may be accessed here, after the barrier
    std::cout << "result=[" << data[0][0] << " "  << data[1][0] << " "
              << data[2][0] << " " << data[3][0] << " " << data[4][0]
              << "]" << std::endl;
    return 0;
}
```

در *SuperGlue* وابستگی‌ها با ثبت کردن دستگیره‌ای که هر وظیفه استفاده می‌کند مشخص می شود. دستگیره‌ها اشیائی هستند که متغییرهای اشتراکی را نشان می‌دهند. در مثال بالا، یک دستگیره به ازای هر قسمت از برداره داده ایجاد شده است.

```
Handle<Options> h[numSlices];
```

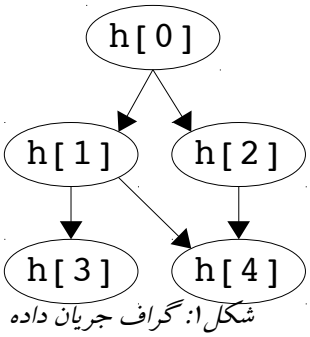
SuperGlue از ارتباط میان دستگیره‌ها و متغییری که از آن محافظت می‌کنند آگاهی ندارد. بنابراین می‌توان از دستگیره‌ها برای نمایش هرچیزی استفاده کرد.

دستگیره ها در موازات زمانی که وظایف متناظر با داده ها، ساخته می‌شوند، ارسال می‌گردند. در سازنده کلاس، وظیفه باید دسترسی خود را به دستگیره و همچنین نوع خواندنی یا نوشتنی بودن داده را مشخص کند. در *ScaleTask* این مساله به شکل زیر انجام می‌پذیرد.

در اینجا، *ReadWriteAdd* کلاس پیش‌فرض نوع دسترسی است که۳ نوع دسترسی متفاوت را تعریف می‌کند.

- *Read* : وظیفه داده را می خواند، چندین وظیفه می توانند در یک زمان، یک داده یکسان را بخوانند.
- *Write* : وظیفه داده را خوانده و بر روی آن می‌نویسد. تنها یک وظیفه می‌تواند بر روی داده در یک زمان بنویسد. دسترسی‌ها باید به ترتیبی که ارسال شده‌اند انجام شوند.
- *Add* : وظیفه می‌تواند داده را خوانده یا نوشته. همچنین ترتیب دسترسی آن با وظایف دیگر که این نوع دسترسی را بر روی داده دارند، می‌تواند دوباره‌چینش[5] یابد.

جریان داده این مثال توسط گراف زیر نمایش داده شده است.



شکل ۱: گراف جریان داده

همان‌گونه که گراف بالا نشان می‌دهد وظیفه h[3] می‌تواند بلافاصله بعد از پایان کار h[1] آغاز شود، در حالی که برای انجام h[4] نیاز است که هر دو وظیفه h[1] و h[2] پایان یافته باشند. این نوع از وابستگی‌ها غیر عادی هستند، به این معنی که این نوع وابستگی‌ها را با تکثیر وظایف به صورت بازگشتی، ممکن نیست به‌دست آورد.

مثال‌های بیشتر

ادامه این راهنما مثال‌هایی هستند که در دایرکتوری *examples/* قرار دارند. این مثال ها در زیر به اختصار شرح داده

5 .Reorder

شده‌اند.

examples/accesstypes

مثالی از نوع دسترسی‌های تعریف شده توسط کاربر است

نوع دسترسی‌های *add*، *write*، *read* با نوع دسترسی جدید *mul* گسترش یافته‌اند. نوع دسترسی جدید *mul* همانند *add* عمل می‌کند. به این معنی که در ترتیب یکسان اما غیر همزمان اتفاق می‌افتد و *add* از *mul* جدا شده است. بنابراین دسترسی‌های نوع داده‌های مختلف به ترتیبی که ارسال شده‌اند اجرا می‌شوند.

examples/customhandle

چگونگی گسترش کلاس *Handle<Options>* را نشان می دهد.

این مثال کلاس دستگیره را با یک عنصر داده و یک متد جدید گسترش داده که همه دستگیره‌ها، آن را با خود خواهند داشت. در این مثال، از این روش برای ذخیره کردن شاخص در آرایه جهانی استفاده شده است و دستگیره با داده‌ای که نمایش می دهد همبسته شده است. بنابراین همه دستگیره ها می‌دانندکه از قسمتی از یک آرایه یکسان در این برنامه محافظت کنند.

examples/dag

نشان می‌دهد که چگونه *SuperGlue* می‌تواندگراف غیر مدور جهت دار(DAG) را برای دیباگ کردن تولید کند.

این مثال با پیاده‌سازی تجزیه چولسکی کاشی کاری شده (وظایف در این مثال تصنعی بوده و کار واقعی انجام نمی‌دهند)DAG تجزیه چولسکی را ایجاد می‌کند. با فعال کردن *features* در ساختار *Options* ، امکان برخی ساماندهی‌ها در *SuperGlue* ایجاد می‌شود و می‌توان درخواست ایجاد فایل *Graphviz*.dot را که وابستگی‌ها را ترسیم می‌کند داد. این ساماندهی‌ها همرا با هزینه خواهند بود و برای فعال بودن در اجرای عادی توصیه نمی‌شود.

examples/dependencies

نشان‌دهنده نحوه ایجاد وظایف با وابستگی در *SuperGlue* است.

مشابه مثال بالا است.

examples/handlewithdata

نشان‌دهنده تدبیری است که در آن نوع داده‌های تعریف شده توسط کاربر شامل دستگیره نیز می شوند

این مثال همچنین دستگیره‌ها را با داده‌ای که نشان می‌دهند، همبسته می‌کند و از استراتژی جایگزین مثال *examples/customhandle* استفاده می‌کند. این استراتژی زمان‌هایی که دستگیره‌ها شامل نوع داده‌های مختلف هستند، بسیار مناسب است.

examples/helloworld

مثالی کوچک از به کاربردن *SuperGlue* است.
مشابه مثال اول این راهنما است.

examples/logging

نشان‌دهنده پشتیبانی ازثبت رخ‌داد است.

این مثال، ثبت رخ‌داد را فعال کرده و فایل رخ‌داد اجرا را ایجاد می‌کند که شامل یک خط به‌ازای وظایف اجرا شده با قالب‌بندی زیر است.

```
NODE THREAD: START_TIME LENGTH NAME
```

هر خط این فایل چیزی شبه به خروجی زیر می‌باشد.

```
 0 4: 1053212 1000434 B
```

که معنی آن به این صورت است که گره ۰ وظیفه‌ای را بر روی ریسمان۴ اجرا کرده، زمان شروع‌آن ۱۰۵۳۲۱۲ بوده و برای۱۰۰۰۴۳۴ سیکل در حال اجرا بوده است وB نام داشته است. واحد زمان گذشته سیکل بوده و مبدا آن اولین وظیفه اجرا شده می‌باشد.
فایل رخ‌داد می‌تواند با استفاده از اسکریپت *scripts/drawsched.py* یا برنامه *tools/viewer* به شکل نمودار مجسم شود.

examples/nbody

پیاده‌سازی شبیه‌سازی *n−body* با استفاده از *SuperGlue* .

نیرو مستقیم میان مجموعه‌ای از ذرات را محاسبه کرده و براساس آن برای تعدادی از زمان ها ذرات را جابجا می‌کند.
در این مثال ثبت رخ‌داد فعال بوده و فایل رخ‌داد ایجاد شده نشان‌دهنده اجرا است. از ویژگی به نام *Contributions* در این مثال استفاده شده است. این ویژگی با۲ برابر کردن میانگیر امکان داشتن دسترسی‌های *add* به یک دستگیره یکسان، برای اجرا در یک لحظه را می‌دهد.
رابط این ویژگی تا به حال به پایداری نرسیده است و در معرض تغییر است.

examples/ pinnedtasks

نشان‌دهنده چگونگی کنترل ریزدانه برای قرار دادن وظایف و اجرای آن‌ها است.

در این مثال، ربودن وظیفه غیرفعال شده است و وظایف به شکل صریح بر روی ریسمان‌های کارگر مشخص شده قرار می‌گیرند. از این رو برای آزمایش کنترل اجرای وظایف استفاده می‌شود.
همچنین در این‌جا، وظایف امکان اجرای بلافاصله بعد از ارسال را ندارند و باید منتظر صدا زدن *start_executing()* باشند.

examples/ subtasks

نشان‌دهنده چگونگی ارسال وظیفه‌ای از وظیفه دیگر است.

ارسال کردن وظایف به شکل پیش‌فرض *thread safe* نبوده اما می‌توان با مشخص کردن آن در ساختار *Options* آن را *thread safe* نمود.
وظایف ایجاد شده با والدین خود ارتباطی ندارند و مستقل هستند. ایجاد وظایف از چندین ریسمان می‌تواند باعث ایجاد بن‌بست شود، اما برای توزیع کردن ارسال وظایف به منظور کارایی می‌تواند مفید باشد. از دیگر کاربردهای آن، وقفه دادن در ایجاد وظایف است. این وقفه از دو لحاظ ممکن است مناسب باشد، نخست این که تعدادی از وظایف ایجاد شده انجام شده، سپس وظیفه جدید اظافه شود تا ازدحام وظایف در یک لحظه وجود نداشته باشد. دیگر این که انتظار زیاد برای پایان یافتن همه وظایف قبل از ایجاد وظایف دیگر از بین می‌رود.

examples/ subtasks

نشان‌دهنده مثالی است که در آن تعداد وابستگی‌ها به آرگومان های آن وابسته است.
در این مثال، تعداد دستگیره‌هایی که وظیفه دسترسی دارد به پارامترهایش وابسته است.

examples/workspace

نشان دهنده چگونگی تخصیص میانگیر کاری ریسمان-محلی است.

SuperGlue به هر ریسمان کارگر اجازه می‌دهد تا مقدار مشخصی از حافظه را که توسط وظایف به عنوان فضای کاری می‌تواند درخواست شود از قبل اختصاص دهد.
این حافظه هر بار که وظیفه‌ای پایان می یابد، توسط ریسمان کارگر اصلاح می‌شود. هدف این کار پرهیز از تخصیص حافظه توسط وظایف است که ممکن است باعث ایجاد مشکل شود.

مثال‌هایی با وابستگی‌های خارجی

دایرکتوری *examples_dep* شامل مثال‌هایی است که به پکیج‌های خارجی وابسته‌اند.

examples_dep/cholesky

تجزیه کاشی کاری شده را با استفاده از *intel MKL* انجام می‌دهد. این مثال بر پایه مثال *SMPSs* ای است که توسط مرکز ابر رایانه بارسلونا توسعه داده شده است. نیاز است *makefile* ویرایش شود تا مسیر صحیح نصب *intel MKL* در آن قرار گیرد.