16o Concurso de Trabajos Estudiantiles, EST 2013

Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía y Física

Algoritmos de Refinamiento Adaptativo de Mallas con Aceleración GPU para Simulaciones Físicas de Gran Escala

Alumno:
Germán Ceballos

Directores:
Oscar Reula
Carlos Bederián

Versión CEST 2013 - Trabajos Finales de Carreras de Grado

Refinamiento Adaptativo de Mallas con Aceleración GPU para Simulaciones Físicas de Gran Escala

Resumen

En este proyecto se diseña e implementa una biblioteca para la resolución de ecuaciones diferenciales en derivadas parciales, utilizando técnicas de Refinamiento Adaptativo de Mallas (AMR) y a la vez tomando ventaja de la aplicación de GPUs en la computación científica. Al momento de iniciar este trabajo, ninguna otra herramienta con soporte para AMR utiliza este tipo de computación para acelerar el refinamiento. En primer lugar se describió el marco teórico con notaciones más acordes y flexibles. A continuación se diseñó e implementó cada uno de los módulos de la biblioteca, para finalmente simular ciertos fenómenos sencillos pero significativos (ecuaciones diferenciales sencillas sobre dominios de gran escala, i.e. mucha resolución). El artículo concluye con los resultados de una numerosa cantidad de pruebas de rendimiento y comparaciones, así como también con un análisis detallado a modo de concluir ventajas y desventajas de este paradigma y posibles extensiones.

Keywords: Adaptive Mesh Refinement, Ecuaciones Diferenciales, GPU, High Performance Computing.

1. Introducción

Una gran parte de las descripciones de fenómenos físicos no son tratables de manera analítica, principalmente debido a la complejidad inherente de los modelos y sus soluciones. Como resultado, nuestro conocimiento actual de dichos fenómenos provienen, mayormente, de la experimentación.

La popularización y difusión de las computadoras durante los últimos años han hecho posible acercarse a las soluciones de estos problemas numéricamente, permitiendo así estudiarlos y realizar predicciones sobre ellos. Cada vez, esta alternativa está resultando más atractiva, y las razones son claras: usualmente, la experimentación (material) es difícil, riesgosa, peligrosa o costosa, y en muchos casos imposible. El estudio a través de métodos numéricos surge entonces como una manera de reducir tanto tiempos como costos en muchas aplicaciones. También hay simulaciones para diseñar objetos, tales como aviones, teléfonos celulares, etc.

Cuando se simula, una de las principales dificultades es lidiar con sus dominios. Muchos de ellos son continuos e infinitos (como el tiempo y el espacio), y por lo tanto intratables para una computadora. Los modelos entonces, deben incorporar algún tipo de discretización de dichos dominios, sacrificando así precisión en los resultados para lograr su tratabilidad. Sumado a esto, pueden no ser compactos, es decir que no poseen ciertas propiedades similares a conjuntos finitos, y por ende en la mayoría de los casos se los *corta* (restringe) y simula sólo una región.

Como los recursos computacionales son siempre finitos y contemporáneos a la era que se está atravesando, su evolución impacta drásticamente en la forma de obtener los resultados y su precisión. De hecho, las decisiones sobre cómo discretizar están absolutamente determinadas por los recursos disponibles, como los procesadores, la memoria y las formas de interconectividad.

El Refinamiento Adaptativo de Mallas (AMR) fue introducido en [2] por Berger y Oliger como una alternativa a los esquemas de resolución fija, en el cual, el dominio infinito (y denso) de un problema es representado por una malla o grilla regular de puntos uniformemente espaciados. Si la distancia entre estos puntos es reducida, la precisión durante la computación aumenta, pero también lo hace el costo de la misma. En situaciones donde las perturbaciones del dominio tienen cierta localidad y evolucionan de manera suave, reducir el espaciamiento de los puntos de la grilla permite analizar el fenómeno con gran detalle. Sin embargo, se aumenta el desperdicio de recursos, ya que las zonas que no tienen interés también estarán representadas con granularidad fina.

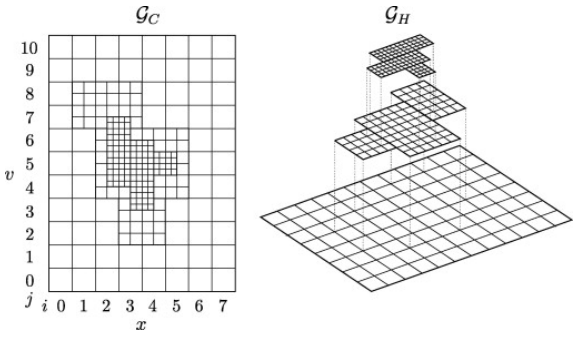


Figura 1: AMR

Además de las estrategias a nivel físico que usualmente son utilizadas para disminuir el tamaño del problema hasta que quepa en el hardware disponible, hay disponibles ciertos métodos numéricos para reducir los requisitos de las aplicaciones.

Aplicar mallas uniformes en dichos sistemas puede resultar en un desempeño realmente desastroso cuando se aumenta la resolución, puesto que genera una enorme sobrecarga de puntos, procesando mucha más información de la requerida. La realidad es que por lo general, sólo se necesitan resoluciones mayores en ciertas regiones específicas de la simulación.

La estrategia presentada por Berger y Oliger intenta solucionar esto, superponiendo grillas de mayor resolución dinámicamente, sólo en aquellas regiones en donde la solución es difícil de aproximar, o se esta perdiendo precisión. Esto se ilustra en la Figura 1.

Las soluciones en todas las grillas serán aproximadas utilizando esquemas de diferencias finitas. Desde el comienzo y durante toda la simulación, se construirá una jerarquía de grillas que será modificada en base a cómo cambian los resultados. Esto es llamado *Refinamiento Adaptativo de Mallas* (AMR), y de este modo introduce una forma completamente nueva de optimizar el rendimiento bajo necesidad de altas resoluciones, ya que requiere muchos menos recursos computacionales que el esquema de densidad fija.

A lo largo de los años y con gran esfuerzo, una gran cantidad de variaciones de la versión original fueron estudiadas y probadas, contribuyendo con la robustez de la técnica. Con el advenimiento de la era de la computación paralela surgió un universo completamente nuevo, tanto para los físicos como para los

programadores, permitiéndoles aventurarse en la adopción de nuevos paradigmas y adaptaciones de sus algoritmos a estas nuevas arquitecturas.

El AMR en paralelo fue presentado varios años después, en varias alternativas, tomando ventaja de las arquitecturas multi-núcleo y multi-nodo. Sin embargo recientemente, la programación GPGPU surgió como una buena opción ante grandes clusters, permitiendo grandes mejoras de rendimiento a un costo bajo. Dado que la tecnología es muy nueva, ha sufrido modificaciones drásticas año a año, pero la comunidad científica en su totalidad está tomando consciencia de su enorme potencial para este tipo de aplicaciones.

Al tiempo en que se realizó este trabajo, existe sólo un framework disponible que implementa AMR y es acelerado por GPUs (presentado en [15]), aunque sólo durante la evolución numérica (cálculo de fluidos), no para acelerar los procesos de regrillado. Se presenta entonces el diseño, implementación y evaluación de un sencillo framework paralelo AMR que toma ventaja de las GPUs para analizar las grillas y refinarlas, además de avanzarlas en tiempo.

2. AMR. Repensado.

El trabajo surge a partir de la investigación de la formación de jets alrededor de agujeros negros binarios. En entornos astrofísicos como este (o similares), en primera instancia se modela el fenómeno físico, (por ejemplo utilizando ecuaciones diferenciales parciales (PDEs)) y luego se utilizan simulaciones para aproximar los resultados y concluir comportamientos.

Para comenzar el desarrollo, se tomó como base un código 2-dimensional utilizado para resolver modelos ecuaciones de onda solitarias que se propagan sin deformarse en medios no lineales, usualmente conocidos como *solitones*. Éstos presentan las mismas características generales que las que esperamos poder tratar con nuestra biblioteca, pero con ecuaciones más sencillas. Los solitones hacen este problema ideal para tratar con AMR. El código tomado es extensamente analizado en [5] y en el más reciente [6].

El problema principal de dicho ejemplo es la utilización una grillas de resolución fija para representar el dominio y las soluciones, y por consecuente la limitación de la precisión a dicho parámetro. El motor del proyecto fue entonces la re-escritura del sistema, utilizando algún framework con soporte para refinamiento adaptativo de mallas (AMR), a modo de comparar los resultados obtenidos con los del código original y mejorar su precisión.

Se realizó un estudio cuidadoso de algunos frameworks con AMR estructurado, como por ejemplo AMRCLAW (LeVeque and Berger), HAD (Liebling), Amrita (Quirk), Boxlib (Bell et.al., LBNL), Chombo (Colella et.al., LBNL), GrACE (Parashar), PARAMESH (NASA Goddard Space Flight Center), SAMRAI (Hornung et.al. LLNL) y GaMER.

Inmediatamente después se detectó que ninguna de estas bibliotecas satisfacía lo buscado, de manera completa. La amplia mayoría fue diseñada e implementada hace más de 10 o 15 años, donde el hardware y las metodologías de programación eran muy distintas.

Adicionalmente, la gran mayoría de ellas están escritas en `Fortran`, muchas en `C++` y algunas en `C`. Si bien muchas cuentan con una versión paralelizada,

pocas hacen uso de los más recientes estándares de comunicación (nuevas implementaciones de MPI, etc), lo que desperdicia poder de procesamiento. Cabe destacar que además, ninguna presenta aceleración del refinamiento con GPUs.

En vista de lo anterior, se tomó la decisión entonces de crear una versión propia: aunque sencilla, contaría con un diseño propio, garantizando la correcta interacción con el código de ejemplo.

3. Descripción del Framework

Aquí se detalla la coincidencia directa entre el marco teórico de la técnica AMR presentado en el Apéndice A y la biblioteca implementada, resumiendo detalles de su organización interna. La Figura 2 muestra el Diagrama de Módulos de la misma, donde se pueden apreciar los módulos funcionales y sus interacciones.

Por otro lado, la Figura 3 muestra las estructuras de datos brindadas por la biblioteca, y su organización a nivel jerárquico.

3.1. Estructuras de Datos

En general, una de las desventajas de trabajar con `C` es la pérdida del uso y manejo de objetos, que brindan un gran poder de abstracción. Sin embargo,

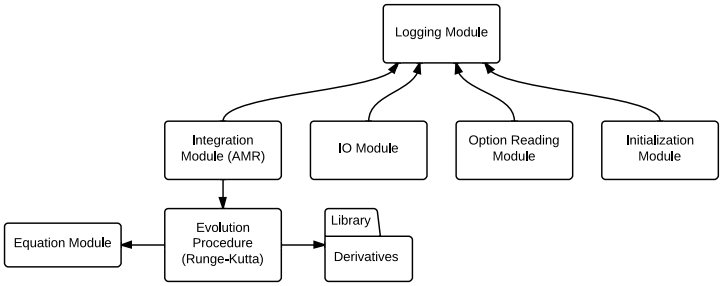


Figura 2: Diagrama de Módulos

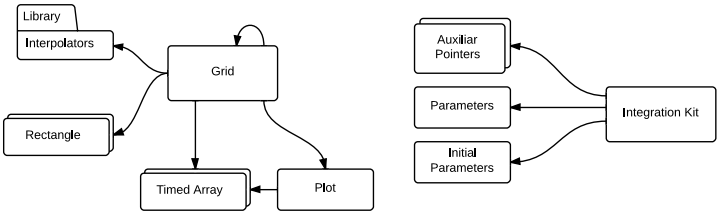


Figura 3: Tipos de Datos

se desistió de usar lenguajes como `C++`para maximizar la usabilidad de la herramienta, ya que introduce muchas dificultades inherentes a la propia sintaxis, dañando la experiencia final de usuario.

Sin embargo, la API que se presenta fue cautelosamente diseñada para no sufrir la pérdida de la orientación a objetos a pesar de utilizar `C`. Todas las APIs se pueden ver en detalle en [1].

3.1.1. Arreglos Temporizados (Timed Arrays)

Para almacenar la información de las grillas (i.e. para una grilla g, la información a guardar es $Im(\check{g})$), es necesario un espacio de lectura-escritura fácilmente indexable y que no genere sobrecarga por empaquetamiento/desempaquetamiento durante las etapas de comunicación.

El código base hacía uso de arreglos contiguos de memoria, dividiendo grillas 2-dimensionales por filas. Durante el rediseño, se introdujo una estructura especial llamada *arreglos temporizados* o `timed arrays`. Estos objetos pueden almacenar datos multidimensionales (es decir conjuntos de $\mathbb{R}^p$) como es el caso de la imagen de cada grilla g, $Im(\check{g})$.

Lo anterior tiene como ventaja la disminución de accesos a memoria cuando los datos son requeridos durante las etapas de evolución y comunicación entre nodos. Representar los datos de las grillas con estos arreglos obliga a convertir los índices del sistema de coordenadas cartesiano ($D1 \times \cdots \times D_n$ points) al sistema de los `timed array` (con rango entre `[0, npoints]`) donde `npoints` $= \#(D_1 \times \cdots \times D_n)$.

Dado que la conversión consiste de meras operaciones algebraicas, el costo de un acceso es relativo a la velocidad del procesador utilizado, lo que en un entorno GPU no es problema.

Además de comportarse como arreglo, también almacena una *marca temporal* o *tiemestamp* dentro de ellos, lo que les otorga el nombre. Esto permite saber cuál es el *tiempo* de los datos almacenados dentro de uno de estos objetos. Dicho tiempo es el valor asignado por la función *time* a una grilla, desde el punto de vista teórico.

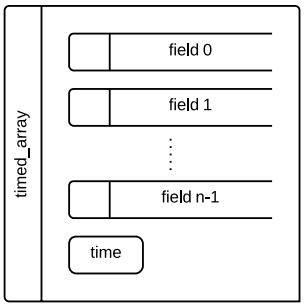


Figura 4: Timed Array

Se tomó la decisión de guardar esta información dentro de los arreglos en lugar de las estructuras de la grilla puesto que, como se estudiará más adelante, para evolucionar grillas muchas veces es necesario utilizar no uno, sino múltiples arreglos temporizados (por ejemplo para campos auxiliares, o derivadas), cada uno con diferente tiempo durante el procedimiento de evolución.

Lo más interesante de estos arreglos es su característica multi-nivel. Esto resulta útil en la resolución de PDEs, dado que permite almacenar múltiples

campos de las ecuaciones en un solo objeto.

La estructura principal consiste básicamente de una región de memoria reservada dinámicamente y algunos contadores. La abstracción es sencilla, limpia y fácil de usar, pero a la vez muy poderosa.

En la Figura 4 se esquematiza la organización interna de esta estructura de datos.

3.1.2. Grillas

Las grillas son el corazón del framework. Son el objeto más complejo en toda la biblioteca, aunque más simples que en otras. Teniendo presente la definición de *grilla* del Apéndice A, se puede ver que en el código, el valor de n está guardado en el campo `dim` de la estructura, $I_1, \ldots, I_n$ son `dim_start[k]` y `dim_end[k]`, donde `k` está en el rango `[0,n-1]`. La resolución r_k es almacenada en `dim_pts[k]`. En cuanto a los intervalos de dominio, sólo es necesario preservar $x_{k,i}$ en `dim_initial[k]` y $x_{k,f}$ en `dim_final[k]` para cada dimensión `k`.

Como se mencionó anteriormente, $Im(\check{g})$ está contenida en el puntero a arreglos temporizados `data`, así como también otra información auxiliar necesaria para los métodos Runge-Kutta (campos no evolucionables).

Dado que no se mantiene ninguna jerarquía explícita de las grillas, fue necesario embeber parte de ésta información en las mismas. Los campos `n_grids`, `refinement`, `father_id`, `child_counter` son utilizados para la creación de procesos con *spawn*, y para la comunicación. Por ejemplo `n_grids` es el número total de grillas en nivel 0.

Sumado a esto, se incluye cierta información relativa a la evolución, como `total_steps`, `initial_time` y `final_time`. Esto fue heredado en alguna medida del código ejemplo, y se puede trabajar en una mejor abstracción para el procedimiento de evolución que incluya esto.

También están presentes algunos campos y banderas para las etapas de estimación de errores y clustering, así como también para la comunicación utilizando `MPI`.

Similarmente a los arreglos temporizados, se cuenta con un constructor y destructor básico, accesores para leer y escribir cada uno de los campos, y algunas operaciones de comunicación. Adicionalmente, se presentan operaciones propias de AMR como las de taggeo, clustering, refinamiento e inyección.

Probablemente una de las más simples pero más importantes partes de la API de las grillas es la función `grid_get_index()` que traduce las coordenadas de los puntos discretos cartesianos a índices dentro del arreglo. Dicha función puede y debe ser configurada por el usuario para garantizar la conversión correcta frente a un cambio en la organización interna de los arreglos temporizados.

3.1.3. Parámetros

Las ecuaciones diferenciales parciales pueden ser utilizadas para describir una gran cantidad de fenómenos, tales como sonido, calor, electrodinámica, fluidos, elasticidad, etc. Todos ellos, aparentemente distintos pueden ser formalizados

idénticamente en términos de PDEs, que demuestra de alguna manera que todos están gobernados por la misma dinámica.

Sin embargo, la ecuación por sí sola no especifica una solución; para obtener una solución única por lo general, es necesario configurar el problema con condiciones extra, tales como **condiciones iniciales**, que prescribe el valor y velocidad de la onda en el tiempo inicial, y las **condiciones de contorno**.

Se proveen dos tipos de estructuras. A pesar de ser y comportarse de manera casi idéntica, definen dos tipos de datos diferentes.

Parámetros Iniciales

Las condiciones iniciales son cruciales cuando se habla del dominio de este tipo de problemas. Un proceso muy importante antes de comenzar con la evolución consiste en *inicializar* los valores de la grilla. El módulo a cargo de establecer dichos valores en tiempo 0 es el *inicializador*, que contiene una función programada por el usuario `initial_data()`.

Uno de las entradas fundamentales para la ejecución de este procedimiento es la estructura `initial_parameters`. Dentro de ella se almacena la lista de parámetros esenciales y sus valores (principalmente constantes), necesarios para la inicialización de la imagen de cada grilla.

El usuario debe llenar el objeto con el nombre y tipo de cada parámetro, respetando el nombre por defecto y la definición de la estructura. Una vez definido, la siguiente etapa es codificar la *función de entrada* correspondiente, en el módulo `input`, de nuevo, sin cambiar ninguna definición. Los miembros de la estructura deberían ser públicos, para permitir su acceso a través de un puntero común y corriente.

La función de entrada lee los valores de los parámetros desde un archivo de configuración externo en tiempo de ejecución. Sus nombres deben preservarse idénticos a los que están en el código. Usar esta estrategia (en lugar de otras como las constantes de compilación, etc.) permiten al usuario cambiar los valores sin necesidad de recompilar ningún módulo.

Al menos una instancia de este objeto debe ser creado durante la inicialización del programa principal, y entregado a la función de entrada, que retornará la misma estructura con los valores leídos desde los archivos de configuración.

Parámetros

De manera similar al anterior, la estructura `parameters` encapsula la información necesaria para la(s) ecuación(es) principal(es). A diferencia de los parámetros iniciales, los valores de este objeto se utilizan múltiples veces durante la ejecución del programa, por ejemplo en las sucesivas llamadas al procedimiento de evolución.

3.1.4. Kits de Integración

Lograr la mayor flexibilidad posible y que el usuario pudiese adaptar la biblioteca a una gran cantidad de fenómenos sin re-compilar es un gran desafío. El software con estas características, especialmente en aplicaciones como las simulaciones físicas, donde un cambio en el problema (ecuaciones, condiciones,

variables, etc.) impacta enormemente en el *diseño ideal* que el programa o la herramienta debería haber tenido.

Desde la perspectiva de la ingeniería del software, este desafío es básicamente pensar qué abstracciones son las mejores para el usuario, y lograr un balance en términos de usabilidad, mantenibilidad, extensionalidad, cohesión, etc. El buen software se nutre de revisiones constantes, por parte del usuario, sobre aquellos aspectos que deberían ser mejorados o pulidos.

En esta sección se introduce una estructura llamada *kit*, o *kit de integración*. Ésta actúa como cápsula, conteniendo tanto un puntero a la implementación del procedimiento de evolución, como a todos sus componentes necesarios para su ejecución. Es importante recordar que un procedimiento de evolución **P** es fundamentalmente una función, y que la definición general presentada con anterioridad es útil para una diversa cantidad de situaciones. Sin embargo, su organización interna y mecánica de operación depende estrictamente del problema a resolver.

Este fue el principal conductor para diseñar y crear los kits. El lector podrá notar una correspondencia biunívoca entre la definición matemática de los procedimientos de evolución, y el módulo de integración que requiere un kit de integración.

Estos objetos permiten preparar cada parte procedimental de la biblioteca para recibir y despachar `kits`, independientemente de su contenido. Si en problemas similares, más herramientas o datos son necesarios, bastará sólo con agregarlos al kit, en lugar de cambiar la definición de los procedimientos.

En los ejemplos provistos, el kit es principalmente una estructura conteniendo punteros a funciones y a estructuras, pero puede tener cualquier otros campos, como constantes o variables.

3.2. Procedimientos

Los procedimientos implementados por el framework son los se comentaron en la Sección 3.1.2. En el trabajo completo [1], se incluye un apartado con un estudio en profundidad de las problemáticas de cada uno durante la implementación, así como también un análisis de su funcionamiento.

Entre ellos se encuentran la inicialización de datos, el *taggeo*, *clustering*, la actualización de bordes, el refinamiento, el *spawning* o engendramiento, el procedimiento de evolución y la inyección/eyección.

Este artículo, sin embargo, se centra en la aceleración de estos algoritmos clásicos utilizando GPUs. Para el estudio detallado sobre la parte procedimental implementada de la técnica AMR, se invita a referirse al trabajo completo.

4. Aceleración GPU

Una vez construida una versión CPU-exclusiva de la biblioteca, la siguiente meta era evaluar el impacto de ciertas modificaciones a los algoritmos para tomar ventaja del paradigma de programación GPGPU, más específicamente, introduciendo la plataforma `CUDA` de `NVIDIA`.

Dicha versión CPU brinda soporte para grillas de cualquier dimensión. Si bien es posible extender este soporte a la versión acelerada por GPU, una especial ventaja es apreciable cuando se trabaja con grillas de 1 a 3 dimensiones gracias

al mecanismo de indexación descrito en la sección anterior. En problemas de estas dimensiones, lo ideal es dividir el dominio inicial (las grillas de nivel cero) tomando como criterio las limitaciones computacionales de cada dispositivo.

Por ejemplo, en grillas 2-dimensionales de gran tamaño, resulta conveniente formar bloques de 32x32 puntos (áreas de 1024 puntos) y distribuir en los procesos regiones de estos bloques directamente proporcionales tanto a la cantidad de multiprocesadores como al número máximo de bloques por SM. De ese modo, todos los nodos aprovecharán al máximo las GPU, balanceando la carga entre todos sus SMs. Es preferible una mayor cantidad de procesos `MPI` maximizando el uso del poder de cómputo, frente a pocos procesos `MPI` que desperdicien tiempo esperando el procesamiento de ciertos bloques.

El objetivo principal del framework concebido es obtener numéricamente soluciones de PDEs usando AMR, en lo que se involucran tanto tareas independientes al problema y relacionadas con la estrategia de refinamiento, como métodos numéricos más dependientes del problema. Cualquier aceleración alcanzada sobre las primeras podrá ser extendida a múltiples problema, mientras que la aceleración de la etapa de evolución numérica genera más selectividad a la hora de reutilización. Se ha identificado, clasificado y paralelizado partes del código en ambas categorías.

4.1. Aceleración AMR

Todas las funciones codificadas en los módulos `gpu_*` operan bajo la modalidad del Algoritmo : copian los datos necesarios al dispositivo, lanzan *kernels*[1] que operan concurrentemente sobre ellos y al finalizar, copian de regreso los resultados. La sincronización de los hilos previo al copiado es fundamental para asegurar que el trabajo se realizó completamente.

Algorithm 0.1: GPU_Template($grid$)

```
{ CopyDataToDevice(grid)
{ LaunchKernels()
{ SyncThreads()
{ CopyDataFromDevice(grid)
```

4.1.1. Taggeo

En la versión CPU, el proceso de marcar aquellos puntos conflictivos en base a algún criterio c se realiza secuencialmente, recorriendo uno a uno todos los puntos. Esto es trivialmente paralelizable, aún cuándo el criterio dependa de múltiples puntos.

Por lo general estos criterios son pasivos, en el sentido de que no cambian los datos de entrada, sólo los leen y calculan un valor de verdad (booleano). Esto

[1] Para más información sobre los *kernels* y otras terminologías referidas a `CUDA`, se invita a revisar la sección sobre dicha plataforma en la tesis original.

permite ahorrar mucho en comunicación, pues no es necesario traer de regreso los valores de la grilla, sólo los valores booleanos.

El siguiente fragmento de código muestra una plantilla de kernel utilizado para grillas 2-dimensionales.

```
__global__ void __gpuTag__ (  int x_points, int y_points,
                              grid_params,
                              bool *flags) {

      int index_x = blockIdx.x * blockDim.x + threadIdx.x;
      int index_y = blockIdx.y * blockDim.y + threadIdx.y;

      if ( (index_x < x_points) && (index_y < y_points) ){
            int index = index_x * y_points + index_y;
            bool result = criteria(grid_params, index);
            flags[index] = result;
      }
}
```

Todo el conjunto de datos variable copiado a la GPU está identificado por `grid_params`. Una vez lanzado un thread que ejecute dicho kernel, su primera actividad es ubicarse en el espacio (a través de `index_x` e `index_y`) y comprobar si ese punto está dentro de los límites de la grilla y si debe ser analizado, ya que pueden darse situaciones donde se ejecutan más threads que puntos en la grilla.

Acto seguido, se calcula el índice global del thread con las fórmulas anteriores. El llamado a `criteria` simboliza en realidad la implementación del criterio de refinamiento seleccionado, en base a los datos `grid_params`. Una vez tomada la decisión según el criterio, se almacena en el arreglo `flags`, que posteriormente será copiado a la CPU.

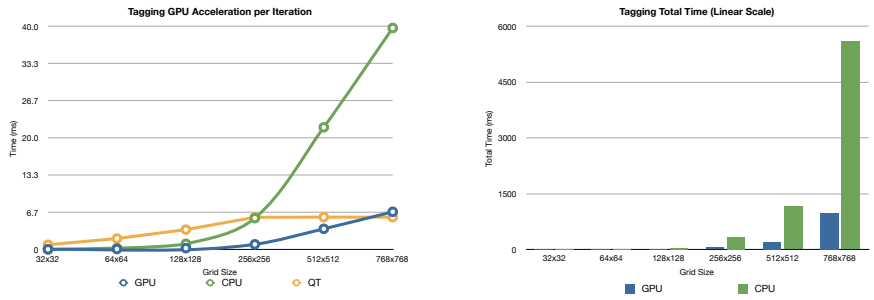
La Figura 5a muestra la diferencia de rendimiento para realizar el mismo trabajo en las versiones con y sin aceleración GPU del algoritmo de taggeo. Cabe destacar que el tiempo y las comparaciones se realizaron tomando la media de los tiempos de todas las iteraciones.

Para ver el impacto en el tiempo total, se puede visitar la Figura 5b, que destaca las diferencias en el tiempo total de ejecución. Las ventajas de usar GPU para estas tareas donde la precisión no está involucrada resultan evidentes.

La ecuación resuelta para estas ejecuciones es la que se describirá en la Sección 5.

4.1.2. Clustering

Una parte más compleja para acelerar con GPUs es la etapa de clustering. El procedimiento aplicado en la versión CPU es recursivo, algo que sólo es factible en placas con `NVIDIA` CC 3.0 o superior (hardware muy moderno), utilizando paralelismo dinámico. Sumado a esto, otro aspecto difícil es la adaptación a GPUs del tipo de datos `rectangle`. Éstos dispositivos resultan muy flexibles en cuanto a poder de cómputo, pero lentos en el manejo de memoria. Se deben pensar mejores alternativas en el algoritmo de clustering para poder sacar alguna ventaja de estos dispositivos.



(a) Una Iteración (b) Total (Escala Lineal)

Figura 5: Aceleración del Taggeo

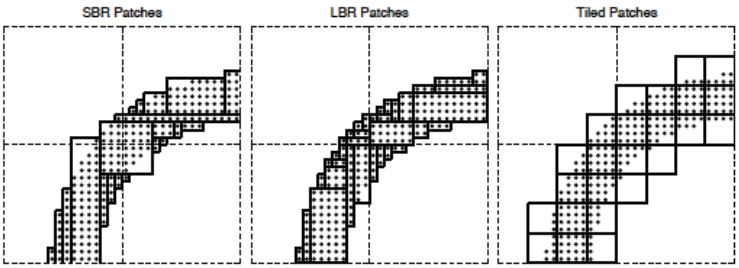


Figura 6: Clustering Paralelo (SBR vs LBR vs Tiled)

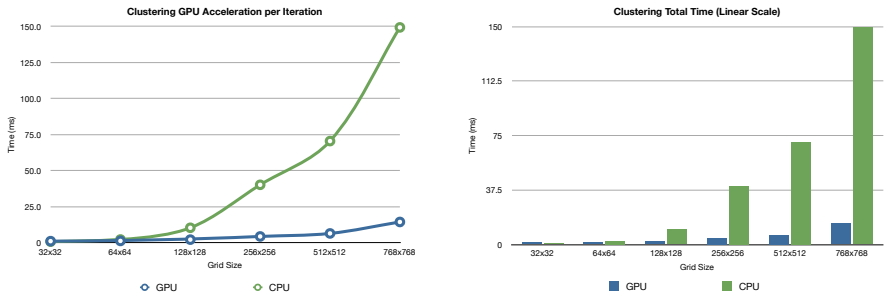
En [20] se evalúan diversas variantes paralelas del algoritmo de clustering clásico secuencial Berger-Rigoutsos (SBR):

Local-BR (LBR): Subdivide la grilla en parches más pequeños, y analiza cada parche concurrentemente corriendo el clásico BR. Basado en Divide&Conquer.

Parallel Global-BR (GBR): Es similar a LBR. Se divide en parches más pequeños y se ejecutan concurrentemente tanto la subdivisión (recortado de parches) junto con la generación de histogramas de cada uno. Después de este proceso, se combinan los resultados de todos los procesos utilizando operaciones como `all-reduce`.

Tiled Clustering (Tiled): Subdivide la grilla original en parches de idéntico tamaño. Luego, corrobora en paralelo si algún punto dentro del parche está marcado para refinamiento, y si es así, se anota para crear una nueva grilla. De lo contrario, es descartado.

La Figura 6 muestra los resultados de correr las tres variantes sobre un conjunto de flags de una grilla analizada.



(a) Una Iteración (b) Total (Escala Lineal)

Figura 7: Aceleración de Clustering Total

La más atractiva es la última. El Tiled Clustering introduce una elegante forma de simplificar las estructuras de datos necesarias: al ser los rectángulos todos de igual área, basta con almacenar sólo un identificador (r_{id}) de ellos para indicar su posterior refinamiento, en lugar de las coordenadas de cada uno. Este indicador es indexable y posible de almacenar en un simple arreglo unidimensional.

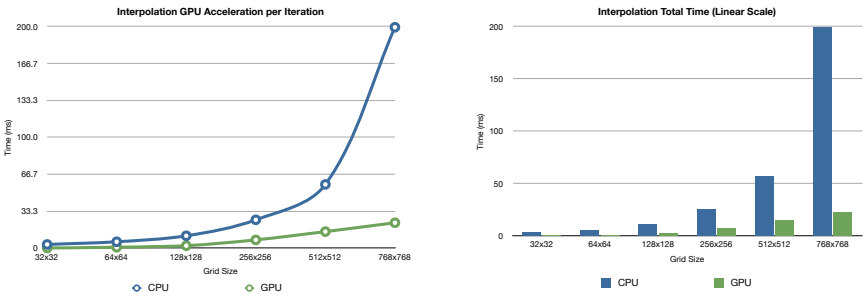
Esta estratega, a pesar de no haber sido diseñada para GPUs, hace el máximo aprovechamiento de los recursos de las placas: permite dividir el problema en bloques equivalentes, y analizar cada punto por separado en un thread para luego calcular un resultado cooperativamente. El resultado es un arreglo de tamaño igual a la cantidad de rectángulos, con valores booleanos verdaderos para aquellos parches que deban refinarse y cumplan con los niveles de eficiencia.

Se obtuvo en general un aumento de velocidad de 10x a 12x utilizando Tiled Clustering (Figura 7a). Si se combinan la etapa de taggeo con la de clustering en la GPU, es posible potenciar el rendimiento dado que se elimina la necesidad de copiar el arreglo de flags desde y hacia la CPU: al finalizar el tagging, las banderas quedan en la placa, y el kernel de clustering directamente lo lee desde la memoria del dispositivo. Las Figura 7b muestra también las diferencias en el tiempo total, según el tamaño de grilla.

4.1.3. Interpolación

Durante el refinado, es necesario tomar regiones de grillas más gruesas e interpolarlas para crear nuevos parches de mayor resolución. Esta es quizás la tarea que demanda mayor cantidad de tiempo, porque no sólo se realiza punto por punto, sino que también se realiza rectángulo a rectángulo, resultantes del clustering.

Una estrategia de paralelización naïve es idéntica a la introducida en los dos anteriores apartados: lanzar un thread por punto, y calcular el valor resultante en base a una cantidad k de puntos, donde k depende del orden del integrador. Sin embargo, esto escala pésimamente sobre k, haciendo la biblioteca no apta para simulaciones de *gran escala*. Mientras mayor es el orden del integrador, no sólo



(a) Una Iteración (b) Escala Lineal

Figura 8: Aceleración de Interpolación Total

más puntos son necesarios, sino que también hay más condicionales presentes dentro de los kernels.

En los interpoladores de 4to orden, hay entre 6 y 8 expresiones condicionales que tienen pésimo rendimiento en los procesadores gráficos, cada uno con hasta 4 condiciones. Si no se es cuidadoso, es probable que el acercamiento naïve sea incluso más lento que la implementación CPU.

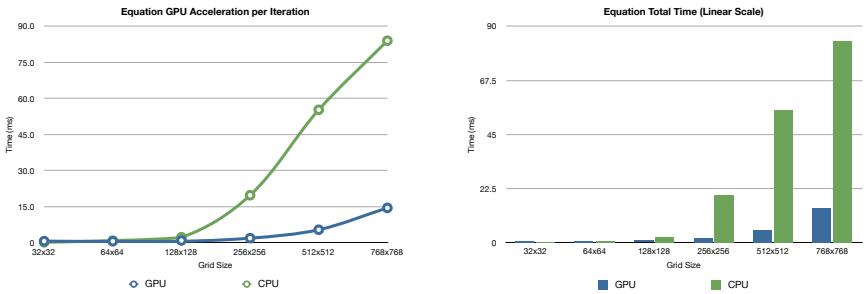
Afortunadamente, las placas gráficas dada su naturaleza de procesamiento gráfico, cuentan con elementos llamados *texturas*, con memoria especial para estos elementos, y ciertas operaciones sobre las texturas que se ejecutan directamente sobre hardware especializado a fin de acelerar el rendimiento.

Una textura plana es básicamente una matriz bidimensional con valores dentro de cada punto (por ejemplo, valores de luz, color, etc.). Entre las operaciones sobre texturas más frecuentes encontramos los filtros (*linear filtering*), los cuales consisten de interpolaciones implementadas sobre hardware.

Gracias a estas herramientas, surge una nueva alternativa de calcular las grillas refinadas. El procedimiento acelerado consiste entonces en construir una textura con la información de la porción gruesa a refinar, invocar la operación del filtrado, y copiar de regreso los datos hacia una nueva grilla de mayor resolución. Esto es enormemente mas rápido, y escala de manera superior a medida que las grillas crecen, dado que se utiliza hardware específico.

A pesar de todo, este método tiene algunas desventajas notorias. Por un lado, el uso de filtros sobre texturas soporta sólo hasta tres dimensiones. Por otro, los valores de las texturas sólo son de precisión simple. En estas circunstancias se recomienda al usuario comparar cuán significativa es la pérdida de precisión en el resultado final entre las versiones CPU y GPU, y optar por alguno de los esquemas.

Las Figuras 8a, 8b, muestran mejoras en el rendimiento alcanzadas en los ejemplos estudiados.



(a) Una Iteración (b) Total (Escala Lineal)

Figura 9: Aceleración de Ecuación Total

4.2. Aceleración Numérica

4.2.1. Ecuación

Las mejoras conseguidas en la resolución de las ecuaciones se debieron a la amplia paralelización de ciclos. Se remplazaron todos los accesos y modificaciones secuenciales en la evolución temporal, por accesos y modificaciones en paralelo, utilizando *threads* y *kernels* provistas por `CUDA`. La mecánica es la misma del *stencil* o esquema .

También se hizo uso de ciertas bibliotecas con integradores acelerados por GPU como las de [21]. Todos estos cambios aceleraron fuertemente el procesamiento de grillas de mediano y gran tamaño, como se puede apreciar en las Figuras 9a y 9b.

5. Ejemplo: Ecuación de Onda 2D Lineal Sobre la Esfera.

La primer ecuación simulada es $\partial_t^2\phi = \Delta\phi$, con $\Delta f = \nabla^2 f$ el laplaciano en coordenadas esféricas. Como el dominio es regular pero no bajo coordenadas cartesianas, se utilizó el esquema de la Figura 10, donde se representa la esfera con un cubo de 6 grillas inter-conectadas.

Cada una de las seis grillas iniciales de nivel 0, $g_0 = (\check{g}_0, t_0), \ldots, g_5 = (\check{g}_5, t_5)$, representa una sección distinta de la esfera. La ecuación es la misma en cada grilla, con diferentes condiciones iniciales

El intervalo de tiempo estudiado es [0,0.5], iniciando todas en 0 ($t_0 = \cdots = t_5$) y con coherencia de los saltos temporales $dt_i = dt_j$. Los intervalos de dominio utilizados fueron $[-1{,}0, 1{,}0]$ para todas las dimensiones y para cada una de las seis grillas.

Se fijó $I^0_{\{x,y\}} = I^1_{\{x,y\}} = \cdots = I^5_{\{x,y\}}$, puesto que son grillas cuadradas formando un cubo. Las ejecuciones se completaron para

$$I \in \{[0, 31], [0, 63], [0, 127], [0, 255], [0, 511]\}$$

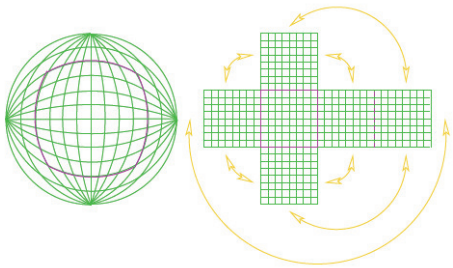


Figura 10: Esfera y Cubo de Grillas

lo que significa que se utilizaron resoluciones con $r \in \{32, 64, 128, 256, 512\}$ puntos.

Para facilitar un poco las computaciones, se fijó también el factor de refinamiento re en 2, duplicando la resolución en cada paso de refinado, pasando por todos los niveles de refinamiento entre 0 y 4.

En cuanto a los parámetros para el refinamiento, se consideró el número mínimo de puntos de una grilla refinada en 50, y la distancia mínima en cada dimensión en 15. El factor determinante del refinado (percentil), al que daremos el nombre de *eficiencia de refinado*, fue fijado a los tres valores $\{0.1, 0.7, 0.9\}$. Lo anterior significa que para refinar un parche, se requiere 10 %, 70 % y 90 % de los puntos taggeados, respectivamente.

A su vez, el criterio de refinado fue basado en el cálculo de la energía en cada punto, i.e. definiendo $c : \mathbb{R}^2 \to \mathbb{B}$, como

$$c(\bar{x}) \quad \doteq \quad \check{c}(\bar{x}) \geq \texttt{threshold}$$

donde

$$\check{c}(\bar{x}) = \frac{\phi_t^2(\bar{x}) + \phi_x(\bar{x}) \cdot up_x(\bar{x}) + \phi_y(\bar{x}) \cdot up_y(\bar{x})}{\phi^2(\bar{x}) \cdot dx^2 \cdot dy^2},$$

El valor de `threshold` fue 0.4.

Luego de múltiples ejecuciones para los conjuntos de parámetros, se realizaron tres comparaciones fundamentales, la primera para demostrar la relación entre incremento de rendimiento y la pérdida de precisión; la segunda clarificando la ganancia en términos de almacenamiento entre resolución fija y AMR. Por último, la tercera mide el impacto de la aceleración GPU en la parte referida al refinamiento.

5.1. Resultados Versión CPU

Aquí se presentan los resultados obtenidos sobre los recursos computacionales que se detallan en el trabajo completo. Se invita al lector a consultar la bibliografía. Las Tablas 1 y 2 muestran los resultados cuantitativos de las primeras dos comparaciones fundamentales mencionadas en el apartado anterior.

eff	Tamaño	Refinamiento	Tiempo	Rendimiento	Precisión
0.1	512x512	0	106s 780ms	-	100 %
	256x256	1	13s 310ms	87.82 %	97 %
	128x128	2	2s 135ms	98.00 %	91 %
	64x64	3	4s 527ms	95.76 %	72 %
	32x32	4	5s 567ms	94.68 %	63 %
0.7	512x512	0	106s 780ms	-	100 %
	256x256	1	39s 665ms	62.76 %	99 %
	128x128	2	17s 848ms	87.29 %	96 %
	64x64	3	4s 864ms	95.44 %	94 %
	32x32	4	2s 785ms	97.39 %	90 %
0.95	512x512	0	106s 780ms	-	100 %
	256x256	1	156s 610ms	-46.75 %	100 %
	128x128	2	36s 982ms	65.37 %	97 %
	64x64	3	4s 558ms	86.37 %	94 %
	32x32	4	4s 050ms	96.21 %	89 %

Cuadro 1: Impacto del refinamiento en el tiempo y precisión.

eff	Tamaño	Refinamiento	Storage (pts)	Storage Gain	Grids	pts/seg
0.1	512x512	0	6,815,744	-	26	63,829
	256x256	1	5,994,234	12.05 %	32	460,990
	128x128	2	4,454,485	34.64 %	42	2,086,817
	64x64	3	1,782,382	73.85 %	62	393,713
	32x32	4	455,908	93.31 %	71	80,307
0.7	512x512	0	6,815,744	-	26	63,829
	256x256	1	2,846,234	58.24 %	42	71,576
	128x128	2	978,974	85.64 %	49	76,196
	64x64	3	432,983	93.65 %	67	89,013
	32x32	4	322,298	95.27 %	80	115,703
0.95	512x512	0	6,815,744	-	26	63,829
	256x256	1	1,726,039	74.68 %	69	11,014
	128x128	2	487,543	92.85 %	77	13,183
	64x64	3	299,864	95.60 %	93	20,597
	32x32	4	279,680	95.89 %	96	69,132

Cuadro 2: Impacto del refinamiento en el almacenamiento.

Se puede apreciar que el valor de eficiencia seleccionado determina por completo la utilizad del esquema, y su cambio impacta de forma drástica en el desempeño. Las Figuras 11a y 11b comparan gráficamente el desempeño de cada ejecución, según el tamaño de grilla y el valor de la eficiencia de los rectángulos.

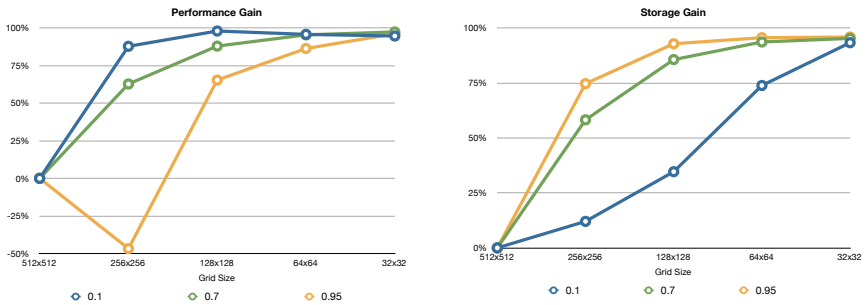
De la Tabla 1, resulta claro que con un valor de eficiencia bajo (0,1), se generan pocas grillas, y por lo tanto habrá menos sobrecarga por generación/-destrucción y comunicación. El incremento de rendimiento es enorme entre la grilla 512x512 y la 256x256 cuando. Si se incrementa demasiado este valor, entonces muchas grillas pequeñas cubrirán las zonas conflictivas, ganando mayor resolución y por ende precisión, en lugares muy puntuales. Sin embargo, la sobrecarga es abismal, como se distingue en la grilla de tamaño 256x256 cuando el valor de la eficiencia es 0.95. Su desempeño negativo muestra que esta ejecución fue 46 % más lenta que la original.

La ganancia de desempeño no es independiente de otras variables, como la **precisión** y el **almacenamiento**. Por ello, no es posible afirmar que el valor 0.1 en la eficiencia es la mejor de las posibilidades.

De la Tabla 2 y la Figura 11bse evidencia que la eficiencia 0.1 es la peor opción para la optimización del almacenamiento durante la ejecución, y esto es esperable, ya que cubrir con parches más grandes, reduce la sobrecarga por generación de grillas mientras que al mismo tiempo cubre espacios innecesarios con resolución alta. También se aprecia que una eficiencia alta (0.95) es la que optimiza de mejor manera, cubriendo con parches más finos y delicados.

Los dos contraste anteriores deben ser bien resueltos para alcanzar un equilibrio en la búsqueda por la optimización. Esto empeora si se incorpora la variabilidad de la precisión según el valor de la eficiencia. Lamentablemente, estas técnicas numéricas pierden precisión cuando se disminuye la cantidad de puntos trabajados, sobre todo cuando se interpolan resultados para inferir otros como en el caso trabajado. El trabajo completo presenta figuras sobre la columna Precisión, clarificando la gravedad de esta situación a medida que se representa el dominio con grillas más pequeñas.

Tratar de analizar manual y exhaustivamente las combinaciones de variables y encontrar una configuración de maximización no es sólo difícil sino que también



(a) Rendimiento (b) Almacenamiento

Figura 11: Resultados según valor de eficiencia.

Tamaño	Refinamiento	Tiempo CPU	Tiempo GPU	GPU Speedup
512x512	0	106s 780ms	78s 890ms	26.12 %
256x256	1	39s 665ms	29s 983ms	24.60 %
128x128	2	12s 848ms	9s 431ms	23.47 %
64x64	3	4s 864ms	4s 184ms	14.72 %
32x32	4	2s 785ms	2s 376ms	14.40 %

Cuadro 3: Impacto de la aceleración GPU en el tiempo (eficiencia mínima 0.7)

es una mala opción. Los espacios a explorar crecen exponencialmente en relación con la cantidad de variables, junto con otras cosas.

Con el fin de no independizar el análisis, en [1] se introduce un estimador (*coeficiente SPP*) que permite tratar estas variables y determinar la mejor variante. Si se aplica a las ejecuciones anteriores, el estimador determina que la mejor opción es la que fija la eficiencia a (0.7).

5.2. Resultados Versión CPU+GPU

Para la eficiencia fijada en (0.7), se realizaron ejecuciones con la aceleración GPU encendida, en las que se obtuvieron los resultados de la Tabla 3. Se presentan sólo las ganancias en tiempo, puesto que el almacenamiento se ha balanceado según al fijar el valor de la eficiencia mínima requerida.

Es claro que la ganancia aumenta a medida que aumenta el tamaño de grilla. En resoluciones bajas, el costo de comunicación es muy representativo en comparación con el de cómputo, lo cual es amortizado en situaciones de alta resolución. Cabe destacar que esta aceleración es sólo durante la etapa de refinamiento y no en la de integración.

6. Conclusiones

Existen muchos estudios que tratan de manera objetiva la técnica AMR, como en [11]. Independientemente de si la estrategia es buena o no comparada con otras, este trabajo estuvo mas bien centrado en mejorar las implementaciones disponibles para exprimir a fondo esta técnica en los nuevos recursos computacionales emergentes.

Se pudo demostrar la ventaja de la paralelización masiva brindada por las GPUs en las tareas de refinamiento que generalmente son dependientes de datos fijos a un determinado tiempo, repetitivas y basadas en conceptos sencillos.

Todo el desarrollo sobre las técnicas de programación para este tipo de plataformas que ha surgido en los últimos 5 años se aplica de manera perfecta y directa, dando como resultado una ganancia enorme frente a un mínimo esfuerzo. Esto nutre a este tipo de herramientas, mejorando su escalamiento frente al crecimiento de los problemas simulados. Como se muestra en los ejemplos, la versión CPU crece de una manera mucho más abrupta al aumentar la resolución del dominio discretizado.

Sin embargo, estas mejoras son todavía muy dependientes de arquitecturas particulares y están muy limitadas. Por ejemplo, hay un cambio enorme en el rendimiento si el problema es de más de tres dimensiones, sin contar que el esfuerzo a nivel programación es muchísimo mayor, pues la optimización todavía no está independizada del hardware.

Si se contempla la aceleración de los métodos numéricos, esto es todavía más notorio. La optimización es significativa, brindando una ganancia muy atractiva. Por el contrario, la capacidad de procesamiento en precisión simple y doble de los dispositivos es determinante en estos casos, causando que el desarrollo del hardware impacte fuertemente en el código producido.

Los miembros de equipo de trabajo son fervientes seguidores de estas plataformas desde su nacimiento, y por ende resultó más que interesante poder realizar este desarrollo y posterior estudio. Al desarrollar esta herramienta, se intenta insistir fuertemente en la incorporación de esta tecnología a la simulación y en especial en técnicas numéricas puesto que las ventajas superan ampliamente el esfuerzo requerido a nivel programación. El beneficio termina siendo grande a la larga, sobre todo cuando es posible mantener bien separada la línea de la *adición* GPU, como en este caso con la habilitación/inhabilitación dinámica de la aceleración.

Se intenta así, incentivar el uso de este paradigma, que se está desarrollando vertiginosamente y de manera casi escalofriante, pero en vista de un prometedor porvenir. Cada vez la unificación de las arquitecturas, la compatibilidad y las flexibilidad de las herramientas es mayor, lo que en un futuro cercano desembocará en un universo enormemente poderoso, alcanzable con códigos sencillos y elegantes.

Se puede revisar una lista detallada de las líneas de trabajos futuros en [1], entre las que se incluye la recuperación de los puntos evolucionados, compatibilidad con otros visualizadores, soporte para otros tipos de fenómenos y el aumentar la paralelización.

El repositorio con el proyecto es público y se a consultar el trabajo completo para obtener una copia.

Apéndice A. Marco Teórico

En este apartado se presentan los fundamentos alrededor del algoritmo propuesto en [2] y en sus subsecuentes variaciones. Las definiciones formales desarrolladas en esta sección son las utilizadas a lo largo del trabajo. Cabe aclarar que aquí se sobre-condiciona la teoría original a grillas rectangulares y uniformemente discretizadas, a modo de simplificar ciertos detalles durante la implementación. Esto es conocido como AMR estructurado.

Para $n \in \mathbb{N}$, se dirá que se *trabaja en n-dimensiones* cuando el dominio del problema sea un subconjunto de $\mathbb{R}^n$.

Conjunto de Índices. Sea $n \in \mathbb{N}$, $k \in [1, n]$ y $n_{k,i}, n_{k,f} \in \mathbb{N} \cup \{0\}$ tales que $n_{k,i} < n_{k,f}$. El *conjunto de índices de la dimensión k* será el intervalo $I_k = [n_{k,i}, n_{k,f}] \subset \mathbb{N} \cup \{0\}$.

Resolución. Sea $I = [n_i, n_f] \subset \mathbb{N}$ un conjunto de índices. Se define r como la *resolución* del conjunto si $r = n_f - n_i$. En caso que I sea el k-ésimo conjunto de índices $I_k, k \in [1, n]$, entonces se le dará el nombre de *k-ésima resolución* o *resolución de la dimensión k.*

Para hacer el dominio tratable, es necesario discretizar un subconjunto continuo de $\mathbb{R}^n$. Para lograr dicha discretización en un determinado problema es necesario analizar cada dimensión por separado.

Intervalo de Dominio. Se define el *k-ésimo intervalo de dominio* a $[x_{k,i}, x_{k,f}] \subset \mathbb{R}$ para dados $x_{k,i} \leq x_{k,f}$ y $k \in [1, n]$, donde $x_{k,i}$ y $x_{k,f}$ son los valores iniciales y finales de la dimensión k en el dominio del problema. De este modo, dado un problema n-dimensional, se representa su dominio como un subconjunto finito (regular/uniforme) de $[x_{1,i}, x_{1,f}] \times [x_{2,i}, x_{2,f}] \times .. \times [x_{n,i}, x_{n,f}]$.

Paso Espacial y Discretización. Sea $I = [n_i, n_f]$ un conjunto de índices con resolución r y $[x_i, x_f]$ un intervalo de domino, se define

$$h = \frac{x_f - x_i}{r - 1}$$

como el *paso espacial del intervalo con respecto a I*. También se denominará como *discretización del intervalo de dominio con respecto a I* al conjunto

$$D = \{x_i, x_i + h, x_i + 2h, ..., x_i + (r-1)h, x_f\} \subset [x_i, x_f].$$

Dichos elementos serán indexados de la siguiente manera:

$$D = \{x_{n_i}, x_{n_{i+1}}, ..., x_{n_{i+(r-2)}}, x_{n_{i+(r-1)}} = x_{n_f}\}$$

Se puede notar que se considera a j como índice de $I = [n_i, n_f]$, entonces $n_i \leq j < n_f$.

Grilla. Sean $n \in \mathbb{N}$. $I_1, \ldots, I_n$ conjuntos de índices con resoluciones $r_1, \ldots, r_n$ respectivamente. Sean $[x_{1,i}, x_{1,f}], \ldots, [x_{n,i}, x_{n,f}]$ intervalos de dominio y $D_1, ..., D_n$ sus respectivas discretizaciones con respecto a $I_1, ..., I_n$. Una *grilla n-dimensional* g será un par $(\check{g}, t)$ donde $\check{g}$ es un mapa

$$\check{g} : D_1 \times ... \times D_n \longrightarrow \mathbb{R}^p.$$

y $t \in \mathbb{R}^{\geq 0}$.

En otras palabras, una grilla g asigna a cada valor determinado t, un valor $\check{g}(\bar{x}) \in \mathbb{R}^p$ para cada $\bar{x} \in \vec{D} = D_1 \times \cdots \times D_n$. Los elementos de $D_1 \times \cdots \times D_n$ serán llamados *puntos de la grilla.*

En términos físicos y para el propósito de este artículo, $\bar{x}$ serán por ejemplo las *coordenadas en el espacio* y t el *tiempo de la grilla.* Ésta es la razón por la cual es usual definir las grillas como restricciones de funciones continuas a dominios discretizados, como $\vec{D}$.

El parámetro p es el número de incógnitas que se intenta resolver dentro de la ecuación diferencial más, eventualmente, algunos campos auxiliares. Por ejemplo en un caso 2D, si se quisiera resolver una ecuación que involucre ϕ, $\partial_x\phi$, $\partial_y\phi$ y $\partial_t\phi$ entonces $p = 4$ y la grilla $(\check{g}, t_0)$ encapsulará toda esta información, devolviendo vectores de la forma

$$\check{g}(x_0, y_0) = (a_0, b_0, c_0, d_0),$$

donde

$$\begin{aligned} a_0 &= \phi(x_0, y_0, t_0), \\ b_0 &= \partial_x\phi(x_0, y_0, t_0), \\ c_0 &= \partial_y\phi(x_0, y_0, t_0), \\ d_0 &= \partial_t\phi(x_0, y_0, t_0). \end{aligned}$$

Notar que se puede definir las grillas sin el concepto del tiempo t y serían tanto matemáticamente, como computacionalmente útiles. Sin embargo, se mantendrá la versión presentada ya que simplifica ciertas de las definiciones subsiguentes.

Va a ser también usual hacer referencia a $\prod_{k=1}^{n} r_k$ como el *número total de puntos* de la grilla, y llamar *parámetros* al conjunto

$$Par_g = \{I_1, ..., I_n, r_1, ..., r_n, [x_{1,i}, x_{1,f}], ..., [x_{n,i}, x_{n,f}]\}.$$

De esta definición resulta claro que una grilla es altamente parametrizable, e involucra una gran cantidad de conceptos.

Anidamiento. Sea $n \in \mathbb{N}$ y sean g, $\tilde{g}$ grillas n-dimensionales con parámetros $I_k^g = [n_{k,i}^g, n_{k,f}^g]$, $I_k^{\tilde{g}} = [n_{k,i}^{\tilde{g}}, n_{k,f}^{\tilde{g}}]$, r_k^g, $r_k^{\tilde{g}}$, $[x_{k,i}^g, x_{k,f}^g]$, $[x_{k,i}^{\tilde{g}}, x_{k,f}^{\tilde{g}}]$ y $D_k^g = \{x_{k,i}^g, \ldots, x_{k,f}^g\}$, $D_k^{\tilde{g}} = \{x_{k,i}^{\tilde{g}}, \ldots, x_{k,f}^{\tilde{g}}\}$ sus discretizaciones de intervalos de dominios respectivamente, para $k \in [1, n]$.

Se dirá que $\tilde{g}$ está *anidada* en g (en símbolos $\tilde{g} \hookrightarrow g$), si se satisface

$$x_{k,i}^g \leq x_{k,i}^{\tilde{g}},$$

$$x_{k,f}^{\tilde{g}} \leq x_{k,f}^g$$

$\forall k \in [1, n]$. Como es necesario que $x_i \leq x_f$ para cada intervalo, se deduce que una grilla está anidada entro de otra, si está contenida en ella.

Refinamiento. Dadas dos grillas n-dimensionales g, g_r como en la definición anterior, con $D_k^g = \{x_{k,i}^g, ..., x_{k,f}^g\}$ y $D_k^{g_r} = \{x_{k,i}^{g_r}, ..., x_{k,f}^{g_r}\}$ sus discretizaciones con respecto a $I_k^g = [n_{k,i}^g, n_{k,f}^g]$, y $I_k^{g_r} = [n_{k,i}^{g_r}, n_{k,f}^{g_r}]$, $\forall k \in [1, n]$. Sea también $re \in \mathbb{N}$.

Se dirá que g_r *refina* a g bajo el factor de refinamiento re ($g_r \boxplus_{re} g$) si

$$g_r \hookrightarrow g, \tag{1}$$
$$x_{k,i}^{g_r}, x_{k,f}^{g_r} \in D_k^g : \;\; x_{k,i}^{g_r} \leq x_{k,f}^{g_r} \tag{2}$$
$$n_{k,i}^{g_r} = re \cdot j \;\mid\; j \in I_k^g \wedge x_{k,i}^{g_r} = x_{k,j}^{g} \tag{3}$$
$$n_{k,f}^{g_r} = re \cdot (j-1) + 1 \;\mid\; j \in I_k^g \wedge x_{k,f}^{g_r} = x_{k,j}^{g} \tag{4}$$

Se puede mostrar (como se ve en [1]) que el paso espacial disminuye proporcionalmente a re. También es importante destacar que muchas grillas pueden refinar a una misma grilla, dependiendo de las coordenadas iniciales y finales. Si $g_1 \boxplus_{re} g$ y $g_2 \boxplus_{re} g$ bajo los mismos parámetros, se considera que $g_1 = g_2$ sólo si $Im(\check{g_1}) = Im(\check{g_2})$ y $time(g_1) = time(g_2)$.

Factor de refinamiento. Una grilla refinada tendrá re más resolución que la que refina bajo factor re. El valor re será llamado *factor de refinamiento*. Una observación válida en este punto es que se considera al factor re constante e idéntico para todas las dimensiones. Esto puede ser trivialmente extendido a $re_1, \ldots, re_n$, constantes para cada dimensión, o incluso siendo $re_i = re_i(t)$ funciones del tiempo u otras variables a modo de hacerlos dinámicos.

Es también importante destacar que si $g_r \boxplus_{re} g$, entonces tanto las coordenadas de inicio como las de fin de g_r coinciden con puntos de la grilla gruesa g. Esto significa que el solapado entre ellas es perfecto dentro de cierta región de puntos del dominio.

Jerarquía de Grillas. El algoritmo AMR usa conjuntos de grillas (*mallas*) anidadas, organizadas jerárquicamente para lograr resolución variable dinámicamente en regiones críticas de las soluciones. Se definirá un *nivel* L_l como un conjunto finito de grillas n-dimensionales $g_1, \ldots, g_l$, donde

$$h_k^{g_1} = h_k^{g_2} = ... = h_k^{g_l}, \qquad \forall k \in [1, n].$$

En ese caso, dicho número será denominado como *paso espacial de nivel* l a lo largo de la dimensión k y denotado h_k^l. Dados niveles $L_0, L_1, \ldots, L_m$, $t_0 \in \mathbb{R}^{\geq 0}$ y un factor de refinamiento $re \in \mathbb{N}$, se define

$$H = (H_g, t_0) = (\{L_0, L_1, ..., L_m\}, t_0),$$

como una *jerarquía de grillas refinadas* en tiempo t_0 si

$$\forall g \in L_i, \;\; \exists \;\; g_c \in L_{i-1} \;\mid\; g \boxplus_{re} g_c, \quad i \in [1, m]$$
$$\forall g \in L_i, \;\; time(g) = t_0 \quad i \in [0, m].$$

En ese caso, L_i será denominado el nivel i, siendo L_0 el más grueso de todos. También se considera que una grilla g está en H si y sólo si $\exists L \in H_g : g \in L$. Por último, el valor re será el factor de refinamiento de la jerarquía de grillas.

Una preocupación general cuando se define una jerarquía es el concepto de *anidamiento correcto*. La amplia mayoría de la teoría desarrollada da lugar a

definiciones más genéricas y flexibles de jerarquías de grillas. Por ejemplo, la extensión de una grilla de nivel m, g_m, podría ser mayor que una o varias grillas de nivel $m-1$. En esas situaciones, se caracteriza a aquellas grillas *correctamente anidadas* si hay suficientes grillas de nivel $m-1$ para cubrir la región de g_m.

Sin embargo, el camino tomado para paralelizar la biblioteca entre múltiples nodos forzó a construir una jerarquía como la que presentamos, donde el anidamiento correcto no es una preocupación, ya que está garantizado por definición.

Intervalo de tiempo. Se define un *intervalo de tiempo* como un intervalo $T = [t_i, t_f]$ con $t_i \leq t_f \in \mathbb{R}^{\geq 0}$. Se hará referencia a t_i, t_f como el tiempo inicial y final respectivamente.

Dado $s \in \mathbb{N}$ y un intervalo de tiempo $T = [t_i, t_f]$, se dará al número determinado por

$$dt_s = \frac{t_f - t_i}{s}$$

el nombre de *paso temporal de T con respecto a s*, donde s es el *número total de pasos*. Análogamente a las definiciones anteriores, se puede definir la *discretización de T con respecto a s* de la siguiente manera:

$$T_s = [t_i, t_f]_s = \{t_k = t_i + k \cdot dt \mid k \in \{0, ..., s\}\}.$$

Algunas veces, se invocará a conjuntos T_s como *intervalos de tiempos s-discretizados*.

Procedimiento de Evolución. Un *procedimiento de evolución* $\mathbf{P}$ será una función

$$\mathbf{P} : \mathcal{G} \times \mathbb{R}^{\geq 0} \to \mathcal{G}.$$

Si $g = \mathbf{P}(g_i, dt)$, entonces se dirá que g es el *avance* de g_i desde $time(g)$ hasta $time(g) + dt$, o simplemente que g es el resultado de avanzar g_i un tiempo dt.

Evolución de Grillas. Dada una grilla g, un procedimiento de evolución $\mathbf{P}$, un número de pasos s y un intervalo de tiempo s-discretizado $[t_i, t_f]_s = \{t_i = t_0, t_1, \ldots, t_s = t_f\}$ tal que $time(g) = t_i$, se define la *evolución de g por $\mathbf{P}$ desde t_i hasta t_f* como la sucesión de grillas

$$(\check{g}_0, t_0) \to \cdots \to (\check{g}_s, t_s),$$

que satisfacen

$$\begin{aligned}
(\check{g}_0, t_0) &= g, \\
(\check{g}_1, t_1) &= \mathbf{P}((\check{g}_0, t_0), dt_s), \\
&\vdots \\
(\check{g}_k, t_k) &= \mathbf{P}((\check{g}_{k-1}, t_{k-1}), dt_s), \\
&\vdots \\
(\check{g}_s, t_s) &= \mathbf{P}(\check{g}_{s-1}, t_{s-1}), dt_s).
\end{aligned}$$

Notar que $g_s = \mathbf{P}(g, s \cdot dt_s)$, pero la evolución por pasos como fue presentada va a ganar especial importancia cuando se revea el algoritmo AMR. Es necesario destacar que esta definición de procedimiento de evolución es suficientemente poderosa en la práctica para representar cualquier transformador de grilla. Como aquí se trabaja con PDEs, usualmente se utilizará métodos Runge-Kutta como integradores, pero esto depende de la naturaleza del problema y de la conveniencia.

A.1. Descripción del Algoritmo

Una vez construida toda esta infraestructura de objetos matemáticos, ya se está en condiciones de continuar con la descripción teórica del algoritmo original y de la versión posteriormente implementada.

A.1.1. AMR Paralelizado

El algoritmo AMR estandarizado se puede estudiar en [3]. Dada una jerarquía de grillas H en tiempo t_i, con nivel inicial L_0, un intervalo de tiempo s-discretizado $[t_i, t_f]_s$, y un procedimiento de evolución $\mathbf{P}$, el procedimiento recursivo garantiza el avance de cada grilla de H desde el tiempo t_i hasta t_f, usando $\mathbf{P}$. El proyecto se basó en todo el desarrollo ya estudiado sobre cómo paralelizar el algoritmo AMR clásico.

Se presentará directamente el paradigma de paralelización utilizado en este trabajo. Se realizaron ligeros cambios a la heurística presentada en [3], principalmente para utilizar herramientas modernas para el desarrollo de software concurrente.

El principal objetivo fue el prototipado de una biblioteca capaz de trabajar ágilmente en clusters de múltiples nodos, cada uno con múltiples núcleos, y eventualmente un par de GPUs. Al mismo tiempo, siempre se priorizó la usabilidad y sencillez de la interface, algo que todos los otros frameworks para diferencias finitas perdieron de algún modo. Plantear un motor poderoso, pero camuflado en una gran experiencia para el usuario es una constante problemática para los programadores, y definitivamente representa una característica muy deseable para el usuario.

Al inicio de esta tarea, se logró visualizar claramente una jerarquía de tres niveles de arquitecturas distintas posibles de aprovechar por separado para mejorar el rendimiento. Cada una con su propio cuello de botella y sus propios problemas. Para los ítems (1) y (2) se debió considerar ciertos problemas de balance de carga.

1. *Comunicaciones Inter-nodo:* Intercambio de información entre nodos de la misma red.

2. *Comunicaciones Intra-nodo:* Intercambio de información dentro de un nodo, entre núcleos.

3. *Comunicaciones Núcleo-GPU :* Intercambio de información dentro de un nodo, entre un núcleo y la GPU.

La plataforma de desarrollo sobre GPUs cuenta con mínima unificación y universalización, consecuentemente, todas las APIs de desarrollo son extremadamente volátiles, haciendo el código muy susceptible al envejecimiento.

Se dio por sentado que (3) representaría el principal desafío, pero definitivamente resultó importante evitar impactos negativos en la biblioteca a causa de los ítems (1) y (2). Se intentó conseguir un diseño capaz de ser indiferente a los problemas de comunicación intra/inter nodo, a modo de ganar transparencia.

En este punto se vislumbró el uso de `MPI` como alternativa. Las últimas implementaciones cuentan con políticas de *scheduling* realmente poderosas e inteligentes, capaces de distribuir (basado en configuraciones) procesos entre *slots* disponibles a lo largo de diferentes nodos, cada uno con sus propios núcleos. Esto es muy útil pues abstrae detalles del hardware disponible al programador. La interface transparente brindada por `MPI` se transfería a nuestro caso, permitiendo lidiar con (1) y (2) como un único conflicto.

A continuación se presenta entonces nuestra variante paralela del algoritmo AMR.

Algorithm .2: AMR$(grid, level, dt)$

$\texttt{Regrid}(grid)$ $\begin{cases} \texttt{TagCriticalPoints}(grid) \\ rectangles \leftarrow \texttt{Cluster}(grid) \end{cases}$
$G_r \leftarrow \emptyset$
for $r \in rectangles$
do $\begin{cases} g_r \leftarrow \texttt{Refine}(grid, r) \\ G_r \leftarrow G_r \cup g_r \\ \texttt{Spawn}(\texttt{AMR}(\texttt{g}_\texttt{r}, \texttt{level} + 1, \texttt{dt}/\texttt{re}^{\texttt{level}})) \end{cases}$

$\mathbf{P}(grid, \frac{dt}{re^{level}})$

for $g_r \in G_r$
do `InjectData`$(grid, g_r)$
`EjectData`$(grid)$
if $(level = 0)$
then `UpdateBoundaries`()

Algorithm .3: INTEGRATE$(grid, [t_i, t_f], s)$

for $i \leftarrow 1$ **to** s
do `AMR`$(grid, 0, dt_s)$

El enfoque de este trabajo fue en mantener una *jerarquía implícita* durante la computación, implementando creación dinámica de procesos bajo demanda provista por el standard `MPI-2`. Todo el algoritmo AMR fue reemplazado por

una recursión sobre las grillas, en lugar de un bucle recursivo sobre las grillas de un nivel fijo de una jerarquía.

El proceso `Regrid` es prácticamente idéntico al original. En él, básicamente se ejecuta un detector de regiones críticas, de acuerdo con un criterio dependiente del problema. Dicho filtro marca o *taggea* todos los puntos en condición crítica, que necesitan ser refinados.

Un procedimiento posterior llamado `Clustering` toma los tags como entrada y aplica ciertas técnicas para agrupar inteligentemente los puntos en objetos que llamaremos `rectángulos` o `cajas`. Esto no es un proceso sencillo. Cada rectángulo marcado se convertirá después en una grilla de mayor resolución, por lo que no es deseable obtener falsos positivos. Debe ser lo suficientemente fino o preciso para descartar aquellas regiones con pocos puntos conflictivos en el interior, así como también detectar puntos de inflexión para dividir grandes rectángulos en más pequeños, maximizando su eficiencia. Se mantuvo la versión más popular del algoritmo Berger-Rigoutsos (BR) de un nivel, presentada en [7] y estudiada posteriormente en [8].

Otra diferencia fundamental que es notable destacar es que se permitió comunicación intranivel entre grillas de nivel 0 únicamente. Esta modalidad de comunicación vertical simplifica en gran medida, y reduce tanto la cantidad de comunicación como las variables requeridas para efectuarla. Esto también significa que, cuando dos grillas del mismo nivel están en contacto (compartiendo un borde), no se envían datos entre ellas. En su lugar, inicialmente se extienden las grillas con una zona fantasma o *ghost zone*, haciéndolas autosuficientes para garantizar la precisión deseada durante la evolución.

La decisión fue tomada después de medir y balancear distintos impactos en el código causados tanto por la comunicación extra como por la computación extra. En un entorno GPU, el principal cuello de botella es el ancho de banda desde y hacia las tarjetas, por lo que resulta preferible aumentar los márgenes de las zonas fantasmas que hacer intercambio de información entre las grillas en contacto.

Si fue preservada la comunicación entre las grillas de nivel 0, puesto que se debe propagar las condiciones de contorno aplicadas en cada iteración. Este esquema vertical, junto con la anterior definición de refinamiento garantizan el anidamiento correcto y simplifican de manera extraordinaria los problemas con la comunicación durante la implementación.

Después de la etapa de clustering, cada rectángulo es ajustado y convertido en una nueva grilla refinada, y luego entregada a un proceso nuevo, que recursivamente aplica el algoritmo AMR hasta que las condiciones de eficiencia son alcanzadas. Luego de la etapa de *engendramiento* o *spawning*, el procedimiento de evolución es aplicado a cada grilla de entrada. Observar que dicha evolución puede ser en efecto realizada en paralelo junto con otras etapas de integración de otras grillas.

A continuación, una vez finalizado el avance temporal, cada proceso debe esperar a todos sus hijos respectivos para sincronizar con el correcto paso temporal. En ese momento, se recibe la información de la solución de aquellos puntos coincidentes en ambos niveles, y se inyectan, reemplazando los valores gruesos con los finos, haciéndolos más precisos.

Para finalizar, se *eyectan* los resultados hacia el proceso padre, alcanzando la

terminación. En caso del nivel 0, sólo se recibe información en lugar de eyectar.

Referencias

[1] G. Ceballos, O. Reula, C. Bederián *Algoritmos de Refinamiento Adaptativo de Mallas con Aceleración GPU para Simulaciones Físicas de Gran Escala.* Tesis de Grado, Facultad de Matemática, Astronomía y Física. 2013.

[2] Marsha Berger, Joseph Oliger. *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations.* Journal of Computational Physics 53 (1): 483-512. 1984.

[3] C. Rendleman, V. Beckner, M. Lijewski, W. Crutchfield, J. Bell. *Parallelization of Structured, Hierarchical Adaptive Mesh Refinement Algorithms.* 1999.

[4] O. Reula, S. Liebling, L. Lehner. *AMR, stability and higher accuracy.* Classical and Quantum Gravity 23: 421. 2006.

[5] F. Carrasco, O. Reula *Solitones en la Esfera: un estudio numérico.* 2011.

[6] F. Carrasco, L. Lehner, R. Myers, O. Reula, A. Singh *Turbulent flows for relativistic conformal fluids in 2+1 dimensions.* 2012.

[7] M. Berger and I. Rigoustos. *An algorithm for point clustering and grid generation.* IEEE. Trans. Sys. Man Cyber., 21 (5):1278–1286. 1991.

[8] O. E. Livne *Clustering on Single Refinement Level: Berger-Rigoustos Algorithm.* UUSCI-2006-001. 2006.

[9] B. Gunney, A. Wissink *A Task-parallel Clustering Algorithm for Structured AMR.* Journal of Parallel and Distributed Computing 66 (11): 1419-1430. 2004.

[10] W. Kim. *Parallel Clustering Algorithms: Survey.* CSC 8530 Parallel Algorithms. 2009.

[11] R. Blikberg, T. Sorevik. *Costs and Savings of Adaptive Mesh Refinement.* 2001.

[12] O. K. Sievert. *MPI Process Swapping: Performance Enhancement for Tightly-coupled Iterative Parallel Applications in Shared Computing Environments.* 2003.

[13] O. K. Siever, H. Casanova. *A Simple MPI Process Swapping Architecture for Iterative Applications.* 2002.

[14] J. Zollweg. *Hybrid Programming with OpenMP and MPI.* 2009.

[15] H. Schieve, Y. Tsai, T. Chiueh. *GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics.* 2009.

[16] T. Boubekeur, C. Schlick. *A Flexible Kernel for Adaptive Mesh Refinement on GPU.* 2009.

[17] D. Luebke. *CUDA: Scalable Parallel Programming for High-Performance Scientific Computing.* 2008.

[18] D. Kirk, W. Hwu. *Programming Massively Parallel Processors.* 2nd. ed. Burlington, USA: Morgan Kaufmann Publishers. 2012.

[19] Rocky Mountain Mathematics Consortium. *General Purpose GPU Computing.* Parallel Numerical Methods for Partial Differential Equations. 2008.

[20] J. Luitjens, M. Berzins. *Scalable parallel regridding algorithms for block-structured adaptive mesh refinement.* Concurrency and Computation: Practice and Experience. John Wiley & Sons. 2000.

[21] R. Manso. *C/C++, CUDA and OpenCL Runge-Kutta Methods.* 2012.

[22] J. Gantz, D. Reinsel. *THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadow s, and Biggest Growth in the Far East.* 2012.

[23] I. Fassi, P. Clauss. *Multifor for Multicore.* 2013.