Informe laboratorio 1 : navier-stoke en un solo núcleo.

El objetivo de este laboratorio fue la implementación de mejoras sobre un código base de una simulación de fluido de navier-stoke para que corra en una CPU.

El primer paso fue utilizar un profiler (en particular utilice gprof y cachegrind) para medir que funciones consumían mayor cantidad de tiempo. Los resultados para el código base fueron que el lin_solve consumía 88% del tiempo, advect 6%, project 5%, mientras que el resto de las funciones estaban por debajo del 1%.

Antes de comenzar a modificar el código se agregaron los flags de compilación -ffast-math (para una versión flexible de las operaciones de punto flotante y mas rápida), -fprefetch-loop-arrays (prefetching de memoria), -funroll-loops (unrolling automático) (En el caso de icc se usaron flags similares). Con estos flags se obtuvo una mejora de alrededor del 10%.

Se observo que dado como estaba definido el macro #define IX(i,j) ((i)+(n+2)*(j)) y el orden de los índices en los loops, se recorrían las matrices en orden column-major lo que perjudica el uso de la cache, y no se ve favorecido por las optimizaciones de prefetching. Se modificaron todas las funciones para comenzar a utilizar el orden row-major.

Se pidió un reporte al compilador para ver que loops se podían vectorizar automáticamente. Se observo que algunos loops no podían ser vectorizados automáticamente debido a posibles dependencias en los datos que el compilador no podía resolver (ej: problema del aliasing). Para resolver este problema, se agrego la keyword **__restrict__** en la signatura **add_source** y **project.** El compilador icc notifico que vectorizo los loops en ambas funciones, mientras que gcc solo pudo vectorizar el de **add_source.** Dada esto, se decidió que **project** se tendría que optimizar manualmente.

Hechas estas modificaciones superficiales, se emprendió en la optimizaciones mas agresivas.
OPTIMIZACION LIN_SOLVE 1 : se cambia :

```
x[IX(j, i)] = (x0[IX(j, i)] + a * (x[IX(j - 1, i)] + x[IX(j + 1, i)] + x[IX(j, i - 1)] + x[IX(j, i + 1)]) ) / c;
```

por :

```
x[IX(j, i)] = res = (x0[IX(j, i)] + a * (res + x[IX(j + 1, i)] + x[IX(j, i - 1)] + x[IX(j, i + 1)]) ) / c;
```

donde res se inicializa a x[IX(0, i)] antes del loop interno.

Estas modificación permitió una reducción importante del tiempo total entre un 25% (icc) y 50% (gcc) obteniendo tiempos que se mantenían casi constantes para diferentes N's.

OPTIMIZACION LIN_SOLVE 2 : usar **sse instrinsics** para obtener una ganancia por medio de la vectorización.
Previamente : se reservo memoria alineada a 16 B para todas las matrices.
La idea es dividir el procesado de cada fila de la matrix en 3 zonas : 2 bordes que no puedo procesar con operaciones vectoriales por que tienen menos de 4 elementos y el centro. Se procesa un vector de 4 elementos utilizando 4 vectores auxiliares (uno para la parte de arriba, abajo y derecha del stencil de x) y uno para x0. Se realizan todas las operaciones vectorialmente excepto la de sumar el elemento izquierdo del stencil. Esta ultimo se hace secuencialmente de izquierda a derecha. Se aprovecha la idea de OPTIMIZACION LIN_SOLVE 1.

OPTIMIZACION PROJECT :
Se vectorizo los loops en project utilizando la misma idea que en OPTIMIZACION LIN_SOLVE 2 de dividir en bordes y centro y vectorizar la mayor cantidad posible de datos. Fue mas sencillo debido a que no había una dependencia de datos como en lin_solve.
Se utilizo la función memset para setear todos los elementos del vector p a 0.

OPTIMIZACION ADVECT :
Para evitar el costo de los ifs que había, se reemplazo (ídem para y):

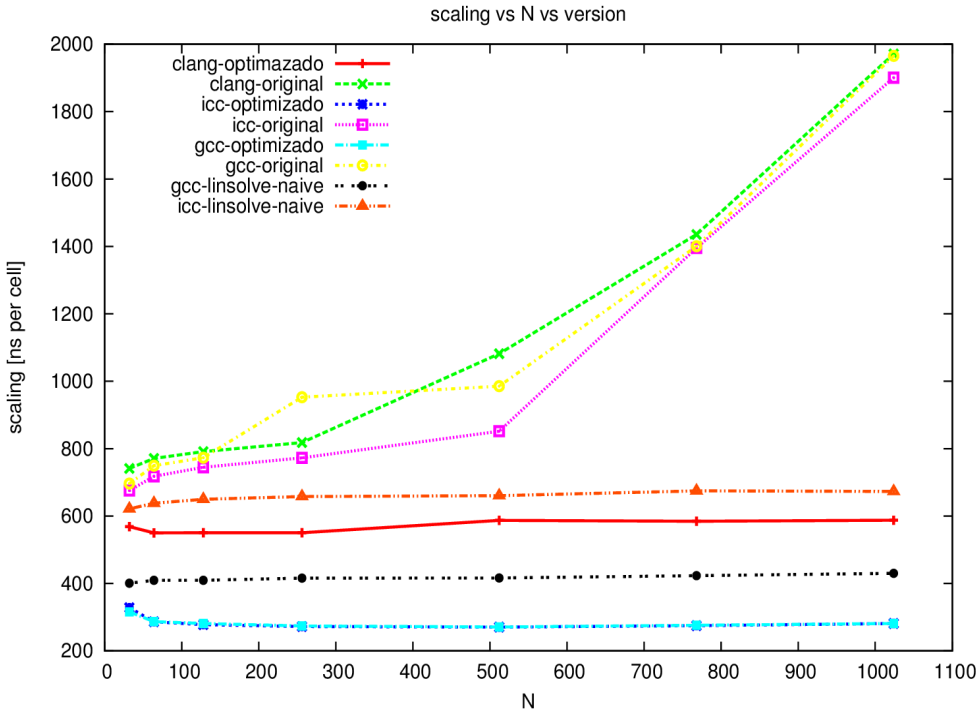
```
 if (x < 0.5f) {
    x = 0.5f;
 } else if (x > n + 0.5f) {
    x = n + 0.5f;
 }
```

por la equivalencia : x = fminf(fmaxf(0.5f, x), n + 0.5f) lo que resulto en una mejora del 3%.
No se vectorizo esta función porque el acceso a la matriz d0 dependía de los valores de u y v en cada celda.

Observaciones :

- Se probo vectorizar add_source pero no acarreo ninguna mejora dado que los compiladores ya la vectorizan automáticamente debido a su simplicidad. Se decidió dejar la versión original, pero quedo comentada la versión vectorizada.
- Dado que set_bound es lineal en n, no consumia mucho tiempo por lo que se prefirió no modificar.
- En lin_solve y project, se trata de acceder a la memoria alineadamente, pero se observa que no se obtienen grandes mejoras.

Gráfico de scaling y Conclusiones



Se gráfico el “scaling” de las versiones original, con solo OPTIMIZACION LIN_SOLVE 1 (para gcc, icc con el sufijo linsolve-naive) y las versiones finales para los compiladores gcc, icc y clang, para valores obtenidos cuando el algoritmo se estabilizaba. Se puede observar que las versiones optimizadas escalan de manera casi constante (en el tiempo que se tarda por celda) a medida que el N crece, lo que es un gran avance en comparación del original.

En general, gcc e icc se comportan muy similar, mientras que clang si bien el algoritmo optimizado escala, se obtienen pocas mejoras para N <= 512, lo que resulta extraño cuando se contrasta con los resultados obtenidos para gcc e icc.

Observando el “speedup” (tiempo original / tiempo de la optimizacion) y el “scaling” obtenido entre las diferentes versiones, se puede concluir que la vectorización fue un el gran valor agregado a la optimización. Aun así, el incremento en el tiempo que tarda por celda el algoritmo original deja a la luz que la forma en que se accedía a la memoria (subsanado utilizando orden row-major y con la OPTIMIZACION LIN_SOLVE 1) a partir de N=512 comienza a ser el factor determinante.