Distribution	# Pts	# Extra Blocks	Time(s)
uniform	2^{18}	70,823	3.16
normal	2^{18}	72,239	3.52
kuzmin	2^{18}	72,917	4.36
line	2^{18}	66,297	3.64
uniform	2^{20}	288,255	13.25
normal	2^{20}	292,580	14.41
kuzmin	2^{20}	292,709	21.34
line	2^{20}	276,124	15.86

Fig. 6. The number of extra 7-byte blocks needed to store triangular Delaunay meshes for various point distributions using our structure and the runtime of our 2D implementation.

address hashing takes prohibitively long. We also require extra memory for the additional map from the label space to the vertices.

7. Experiments

We report experiments on a Pentium 4, 2.4GHz system, running RedHat Linux Kernel 2.4.18, GNU C/C++ compiler version 3.0.1. For all geometric operations (lineside, planeside, incircle, and insphere tests) we use Shewchuk's adaptive precision geometric predicates.[23] We use single-precision floating-point numbers to represent the coordinates. For every problem setting and size the results of our experiments were very consistent over multiple runs. Therefore we do not report ranges of results for identical runs.

7.1. *2D Delaunay:*

We tested our 2D implementation on data drawn from several distributions to assess its memory needs for non-uniform data sets. We ran tests on the following distributions: Uniformly random, normal, kuzmin, and a line singularity. Details on these distributions can be found in Ref. [45]. In Figure 6 we report the number of extra (overflow) 7-byte blocks used to store Delaunay meshes of various point distributions and the runtime of our implementation. It can be seen that the runtime varies by about 40% while the number of extra blocks varies by about 10%. Furthermore the number of extra blocks used comes to only about 28% of the number of default blocks needed, which is one per vertex. In our experiments we set the number of extra blocks available to 35% of the number of default blocks. The extra blocks therefore fill to about 80% of capacity. Given this setting, the total space we require for the mesh is 1.35×7 bytes/vertex, which is 4.725 bytes/triangle.

Next, we compare runtime and memory usage of our implementation to Shewchuk's Triangle[24] code which is the most efficient code reported by Boissonnat et. al..[27] In Figure 7 we report the runtime of our (incremental) code vs. Triangle's divide-and-conquer and its incremental implementation. We report the total