How fast are hash functions?

D. J. Bernstein
University of Illinois at Chicago

NSF ITR–0716498

in cooperation with
IST–2002–507932 ECRYPT

Depends on the volume
of data being hashed.
For small data volumes
can often see big costs for
finalization, block padding, etc.

Depends on the CPU.
e.g. hashing 1000 bytes
with OpenSSL SHA–512:
118366 cycles on a Pentium 3,
16822 cycles on an Athlon 64.

Depends on the hash function.

What is the fastest function
(given a CPU and data volume)?

Depends on the hash function.

What is the fastest function
(given a CPU and data volume)?
“Surely a broken function.”

Depends on the hash function.

What is the fastest function
(given a CPU and data volume)?
“Surely a broken function.”

What is the fastest function
that hasn’t been broken?

Depends on the hash function.

What is the fastest function
(given a CPU and data volume)?
“Surely a broken function.”

What is the fastest function
that hasn’t been broken?

What is the fastest function
for which half the rounds
haven’t been broken?

Experience shows
that slower functions
attract much less interest.

The fastest functions
are the most tempting
cryptanalytic targets.
If they survive scrutiny
then they are favored by users.

e.g.: AES selected Rijndael;
eSTREAM SW selected HC-128,
Rabbit, Salsa20/12, Sosemanuk.

Does hash speed really matter?

Usually it doesn't!
Most computers aren't flooded with cryptographic operations.
Can afford many more rounds.

Some computers *are* flooded, but is hashing the real issue?

Does hash speed really matter?

Usually it doesn't!
Most computers aren't flooded
with cryptographic operations.
Can afford many more rounds.

Some computers *are* flooded,
but is hashing the real issue?

"Yes: I'm using HMAC-MD5
on every packet, including
denial-of-service forgeries.
I can't afford HMAC-SHA-256!"

That's an obsolete application.
We have faster non-hash MACs
that inspire more confidence.

Better example:
"This computer is busy
verifying public-key signatures."

My favorite example:
Internet DNS security
(currently nonexistent)
would need DNS caches to
verify signatures on every packet.
These caches are centralized,
hard to split, and often
very heavily loaded.

Hashing, even for short packets,
can easily dominate the time
for signature verification!

Measuring what users will see

Look back at 1999.03 Bassham "NIST's Efficiency Testing for Round1 AES Candidates."

Remember CRYPTON?
Fastest cipher in NIST's tables!
Fastest cipher in NIST's graphs!
720 cycles key setup.
669 cycles encrypt.

NIST's numbers for Rijndael:
6787 cycles key setup.
809 cycles encrypt.

Quite different figures in
1999 Schneier–Kelsey–Whiting–
Wagner–Hall–Ferguson
"AES performance comparisons."

Faster CRYPTON encryption:
955 cycles key setup.
345 cycles encrypt.

But *much* faster Rijndael:
850 cycles key setup.
291 cycles encrypt.

Users who care about speed will use the faster Rijndael software, not the painfully slow software that NIST tested.

So the painfully slow software was discarded, and NIST's tests were disregarded.

1999.10 NIST: "CRYPTON is ... slower than either Rijndael or Twofish on most platforms."

1999 Schneier–Kelsey–Whiting–
Wagner–Hall–Ferguson:

“Performance is only important
in assembly language. . . .
Any application which has
speed as a requirement
will code the encryption algorithm
in assembly. . . . Optimized
assembly implementations of AES
will be available on the Internet.
If performance is critical,
it will be in assembly.”

I still see some speed papers saying that a "fair comparison" of algorithms means a comparison of speeds of novice student implementations in Java.

That's "fair"? No. It's idiotic. Or dishonest: "We couldn't compete, so we rewrote the competition to slow it down."

Benchmarks, just like users, should focus on the fastest implementations available. Designers with slow software should try to speed it up.

The importance of automation

During the AES competition,
Biham and Knudsen proposed
a $2\times$ security margin.

NIST decided on $(1+\epsilon)\times$.
But NIST refused to consider
reduced-round Serpent etc.

"Changing the number of rounds
would impact the large amount
of performance analysis from
Rounds 1 and 2. All performance
data for the modified algorithm
would need to be either estimated
or performed again."

2004: eSTREAM called for stream-cipher submissions implementing a particular API.
Received > 30 submissions from 97 cryptographers.

De Cannière published software to time the submissions.

Software has been run on dozens of different CPUs.

Designers have seen results, provided faster implementations of the submissions.
Timings were easily updated.

2006, joint work with Lange:
eBATS (ECRYPT Benchmarking
of Asymmetric Systems).
DH; signatures; encryption.

New benchmarking software:
much more data collected,
improved portability, etc.

As in eSTREAM benchmarking,
anyone can submit new software
or improve existing software.
> 20 submissions so far,
providing > 100 pub-key systems.
All submissions are then
measured on many computers.

Today's announcement: eBASH (ECRYPT Benchmarking of All Submitted Hashes).

Anyone can submit a new hash function or a new implementation of an old function. We'll measure all the software on many CPUs.

Benchmarking software has been further improved: e.g., automatic ABI stratification, separating 32-bit compilers from 64-bit compilers on amd64 etc.

Suggestions? Talk to us!

What eBASH does

Currently we're measuring
time to hash 0 bytes,
time to hash 1 byte,
time to hash 2 bytes, ...
time to hash 8192 bytes.

Aligned input and output.
Typical for speed-oriented
applications but maybe we
should also measure unaligned.

Currently 1 core, 1 thread.
No hint of speedups from
massively parallel computation
of, e.g., tree-structured hashes.

Measurement machines run
Linux, BSD, Solaris, etc.
on a wide variety of CPUs.
Largest machine contributor:
NMI Build and Test Laboratory at
the University of Wisconsin.

We're experimenting with ARMs
but haven't included any yet.

Currently no 8-bit CPUs.

No FPGAs. No ASICs.
Will hardware benchmarking
be stuck forever in the dark ages?

For each hash function, currently trying 796 combinations of C compilers + compiler options.

Measurements of the function use the compiler + option that hashes 1536 bytes most efficiently.

“Have to write in C?”
No; can use C++ or asm.

Need other options? Tell us!

eBASH API examples

The eBASH software
has two files that measure
the OpenSSL SHA512 software.

`hash/sha512/openssl/hash.c`:

```
#include <openssl/sha.h>
void crypto_hash_sha512_openssl(
  unsigned char *out,
  const unsigned char *in,
  unsigned long long inlen)
{
  SHA512(in,inlen,out);
}
```

`hash/sha512/openssl/hash.h`:

```
#include <openssl/opensslv.h>
#define CRYPTO_hash_\
  sha512_openssl_VERSION \
  OPENSSL_VERSION_TEXT
#define CRYPTO_hash_\
  sha512_openssl_BYTES \
  64
```

Version is optional;
copied to database of timings.

Integrating Whirlpool:

`hash/whirlpool/ref`
has two files copied from
Whirlpool reference code;
`hash.h` defining `BYTES` as 64;
and an easy 19-line `hash.c`
built from the lower-level
functions in the reference code.

`hash/whirlpool/checksum`
(also optional) contains an
expected hash output.
Any compiled code that
fails to produce this output
is left out of the measurements.

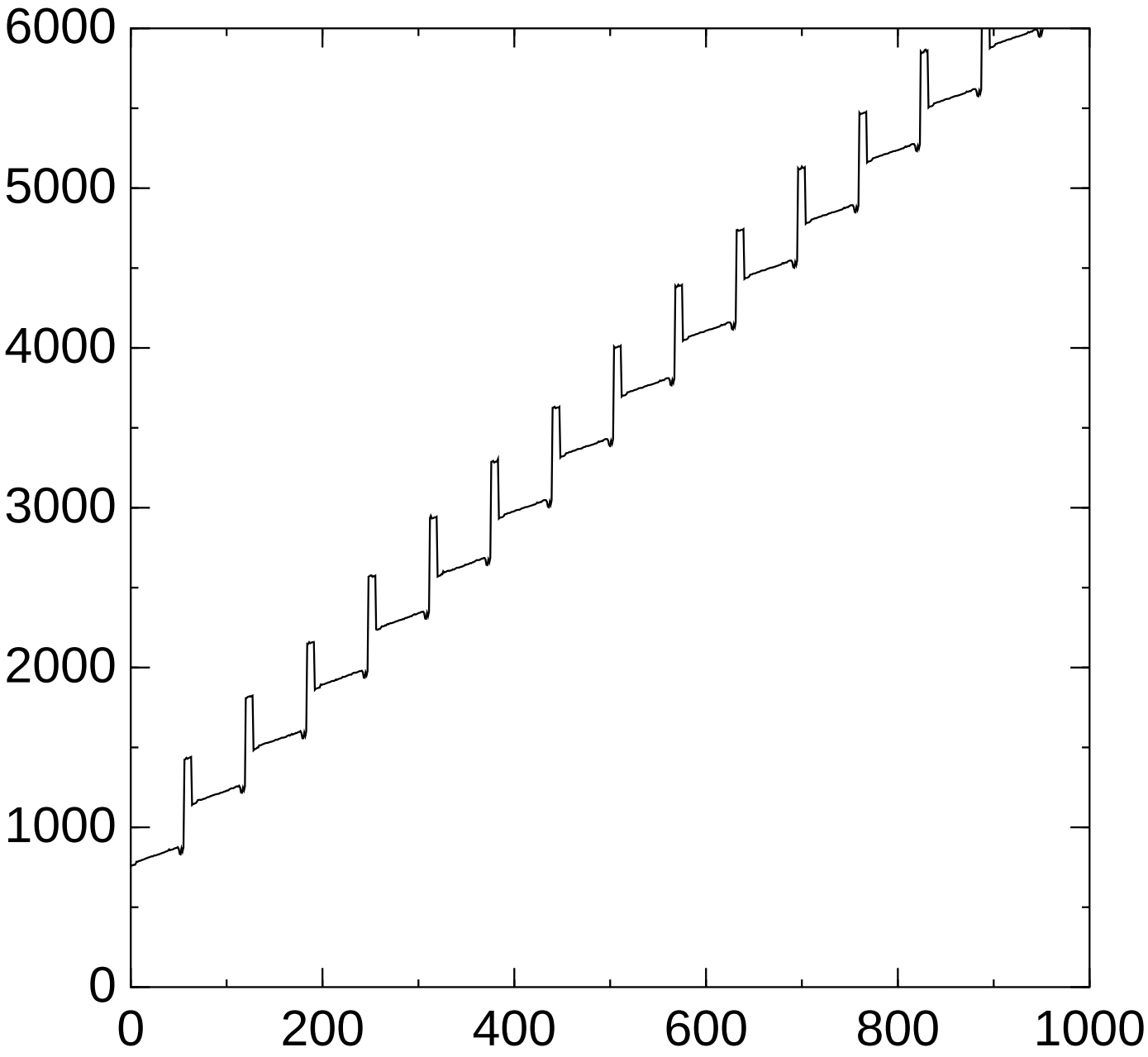
Examples of easy graphs to draw

WARNING: These graphs currently include only a few hash-function implementations.

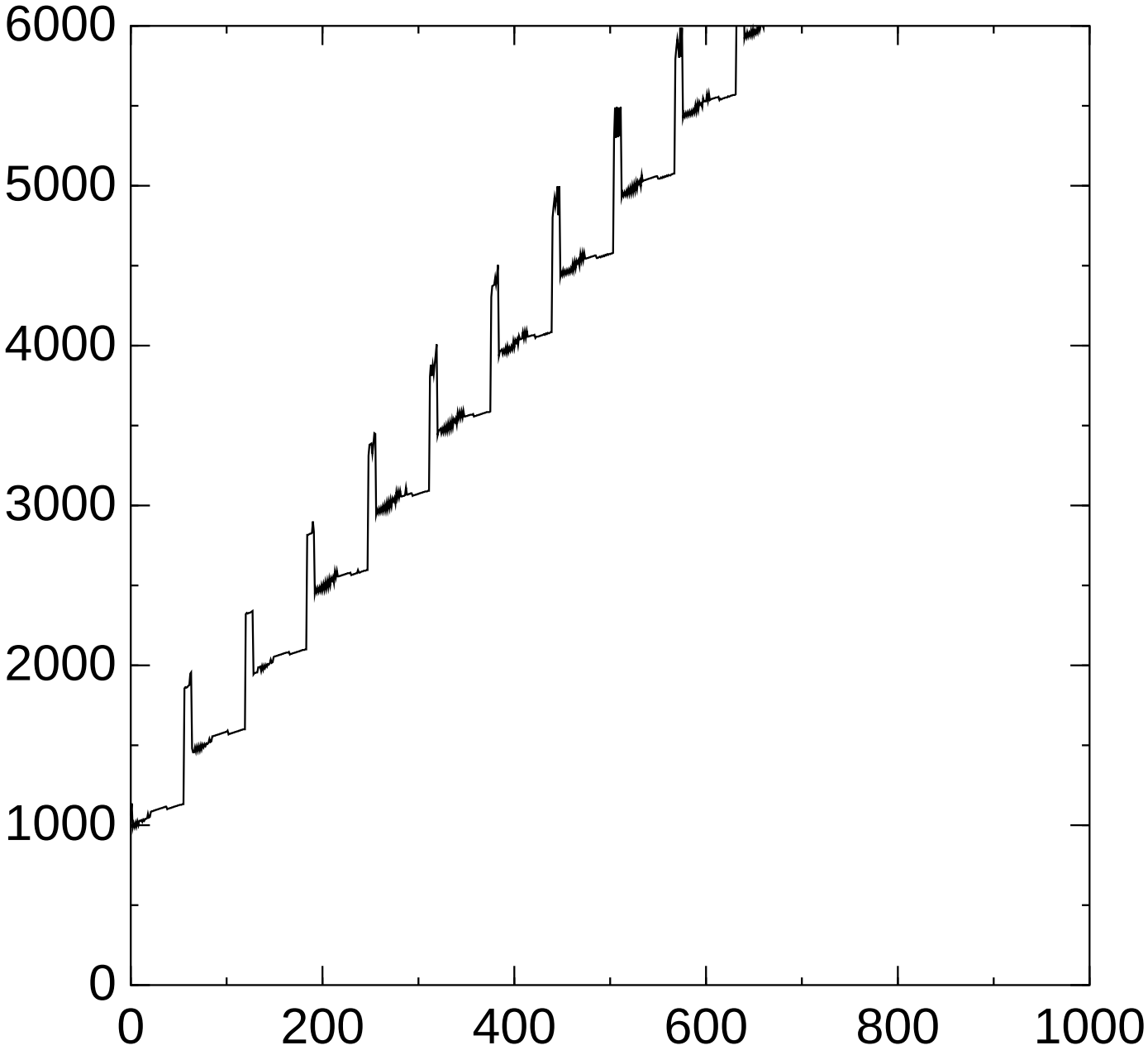
If you have a faster implementation, sorry—it isn't in these graphs. Advertise it by submitting it to eBASH!

We'll easily update benchmarks, graphs, etc.

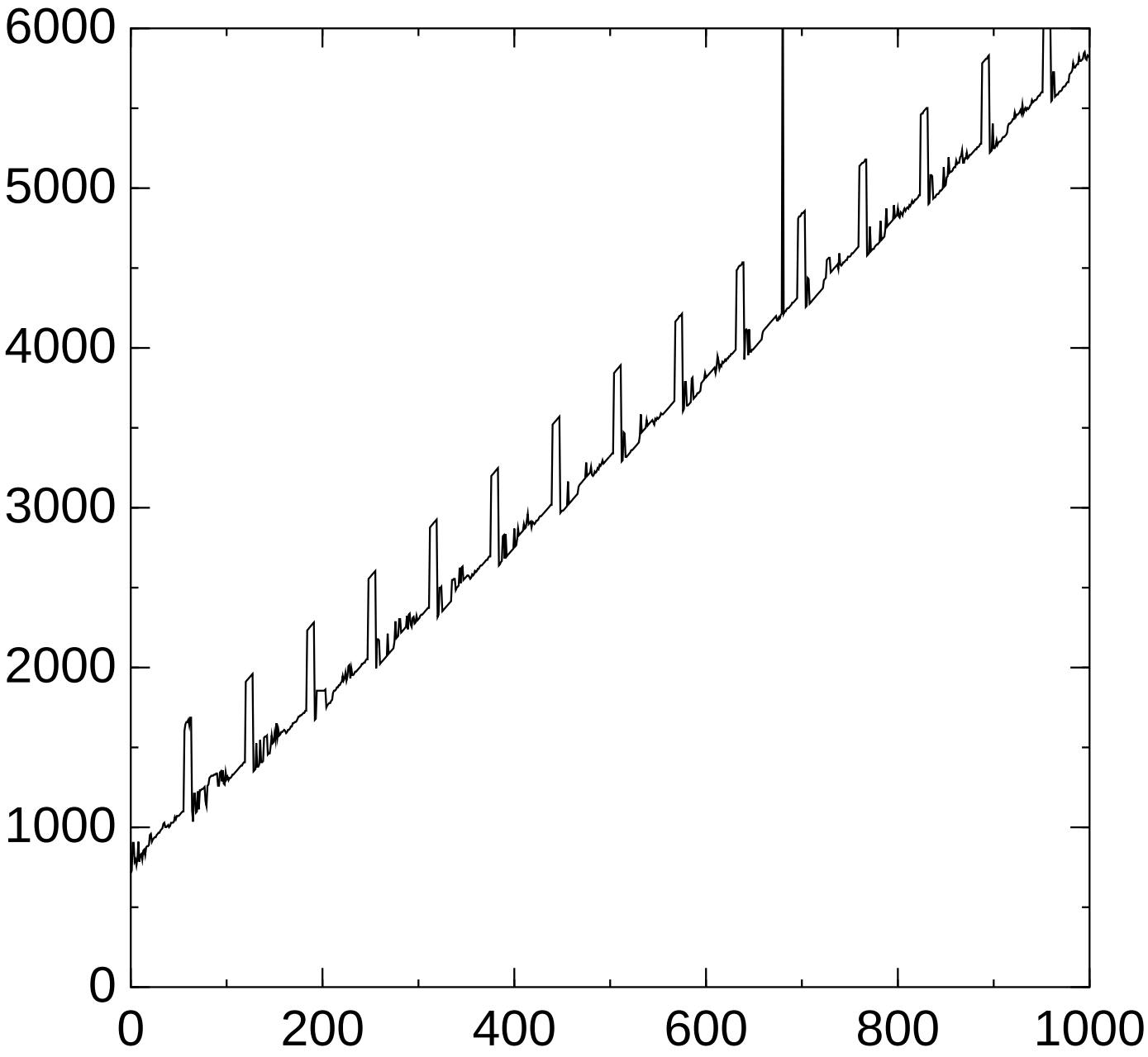
MD5 cycles on `orpheus`,
x86 architecture, Pentium 3 672:



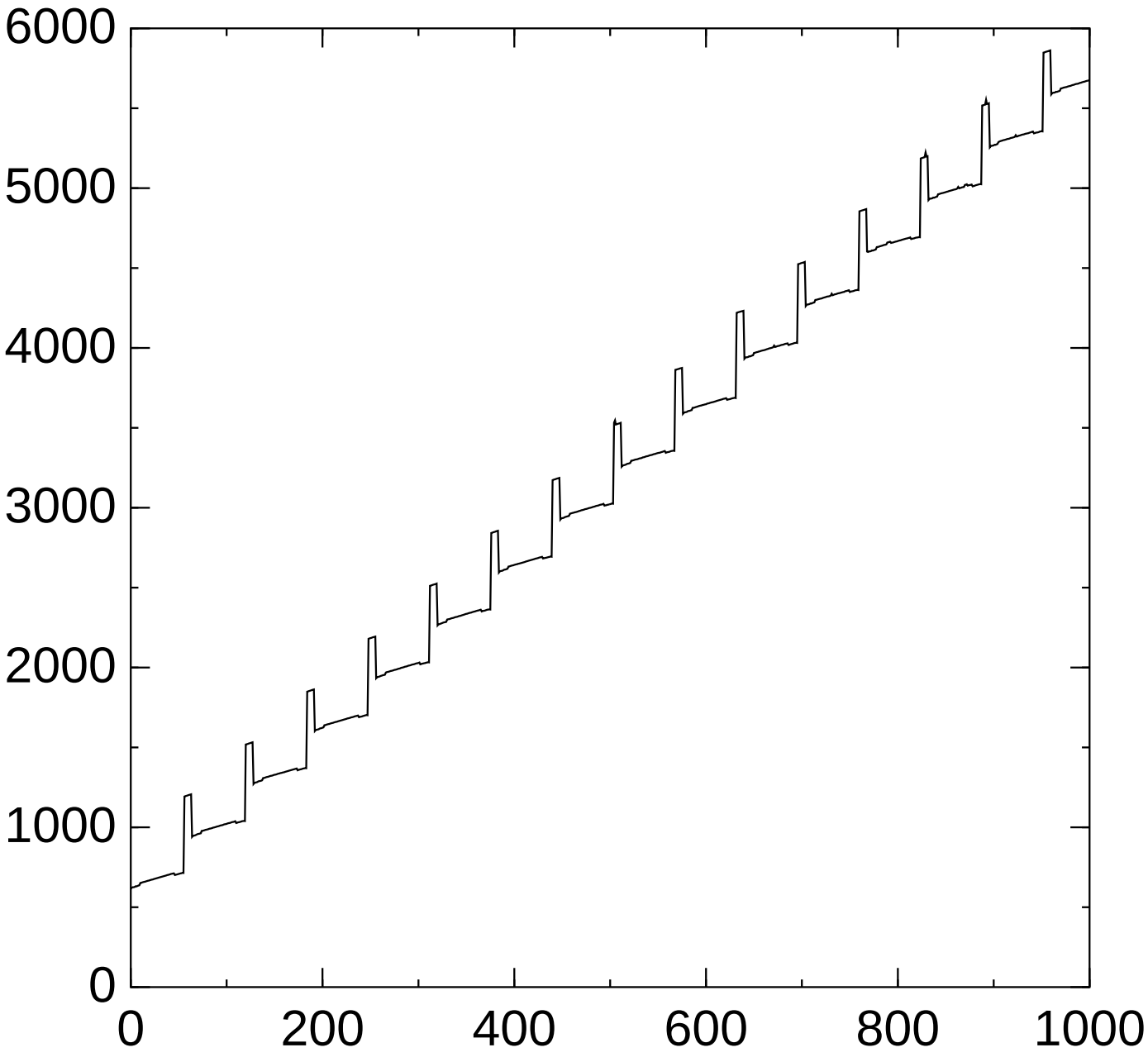
MD5 cycles on `fireball`,
x86 arch, Pentium 4 f12:



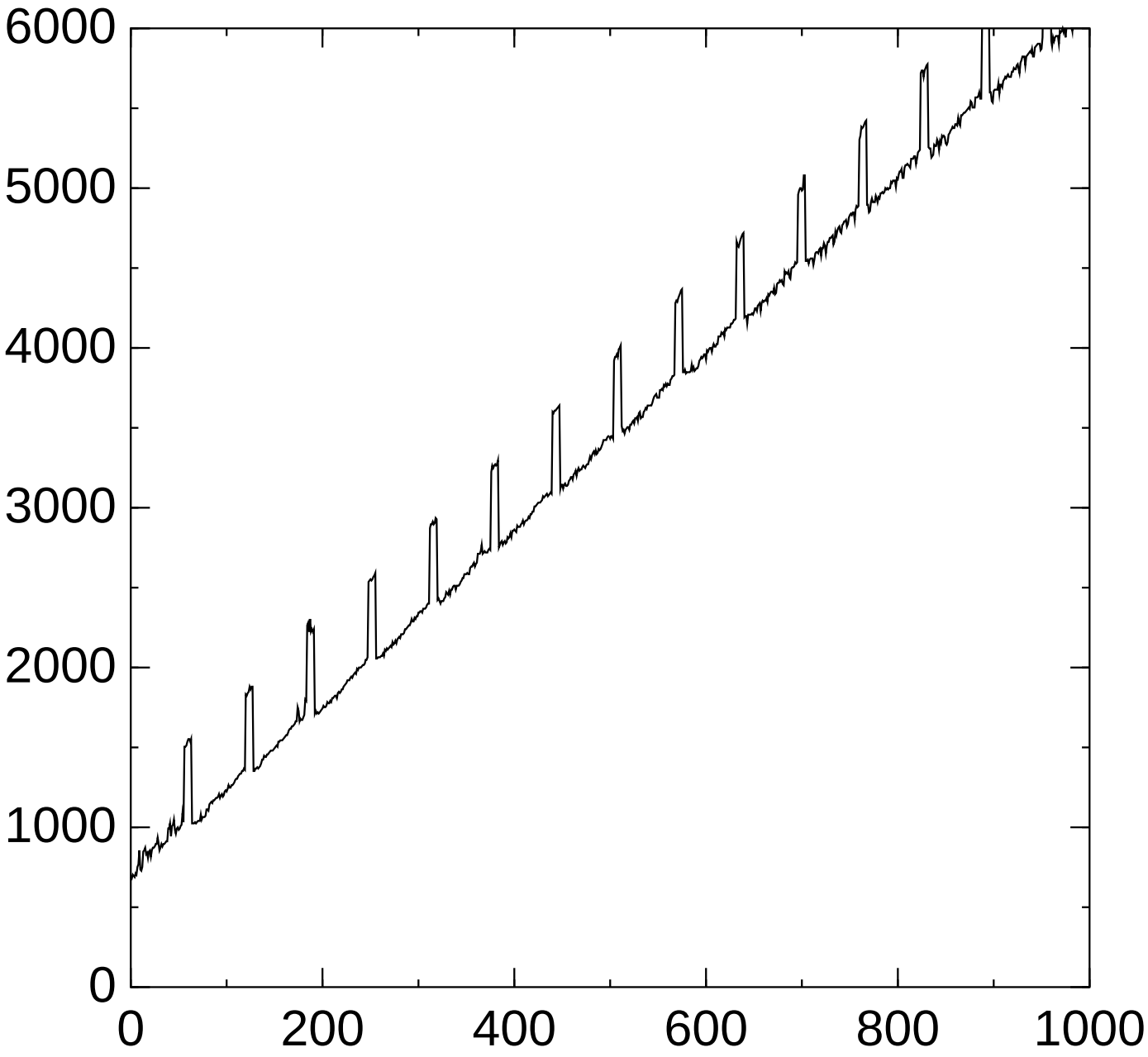
MD5 cycles on `nmi-0056`, x86 arch, Xeon f41:



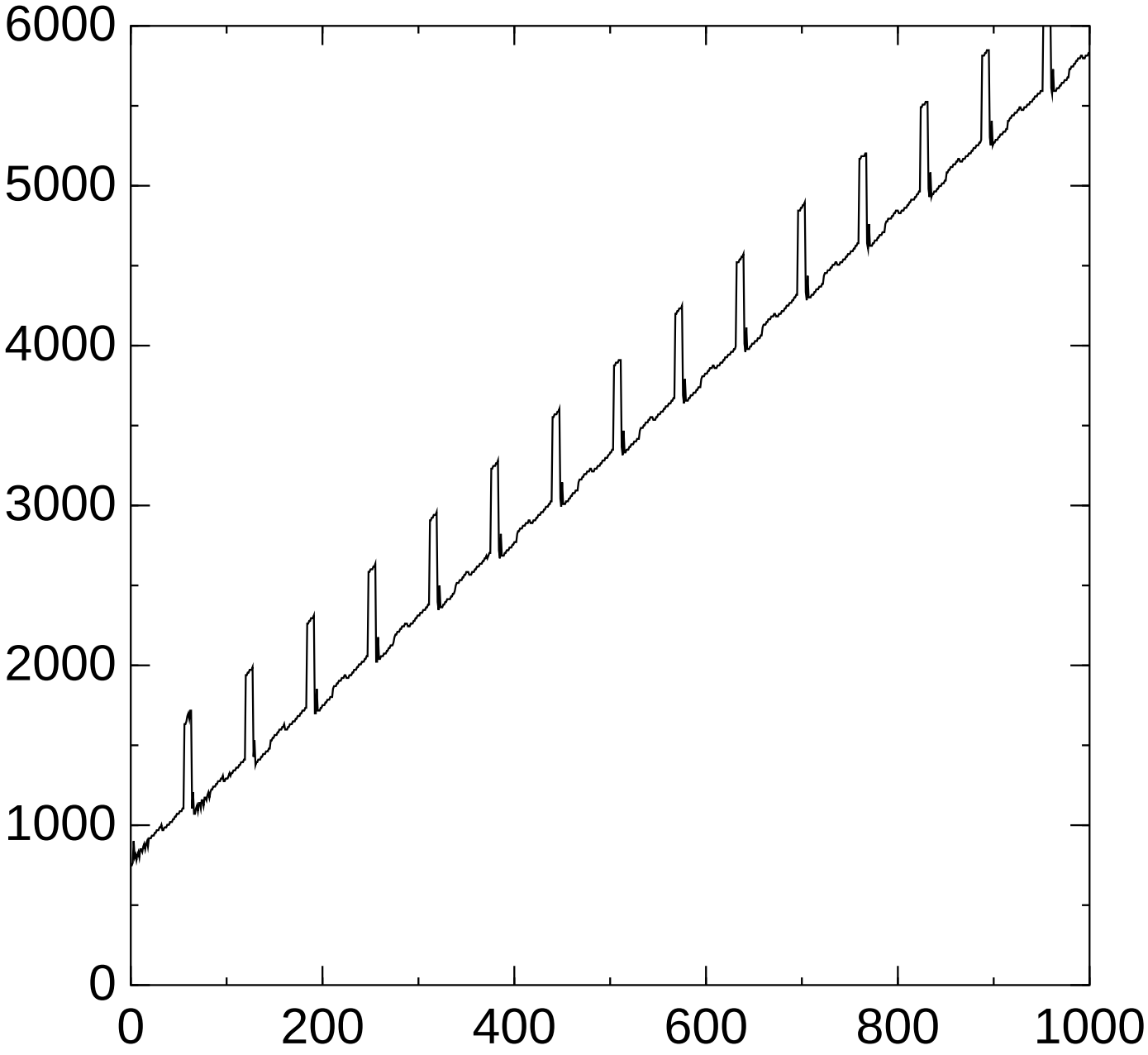
MD5 cycles on `thoth`, x86 arch, Athlon:



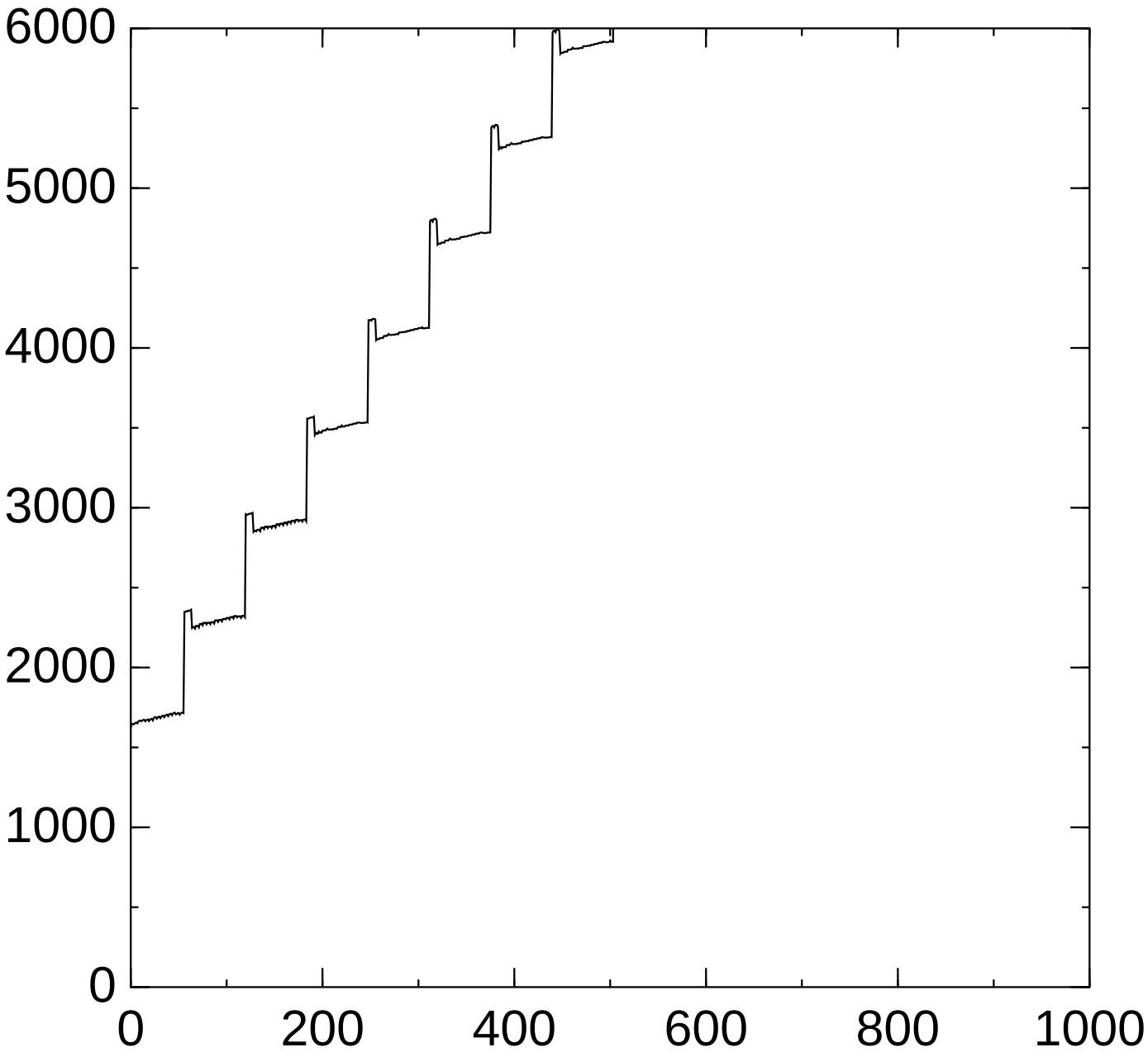
MD5 cycles on `katana`,
x86 arch, Core 2 Duo:



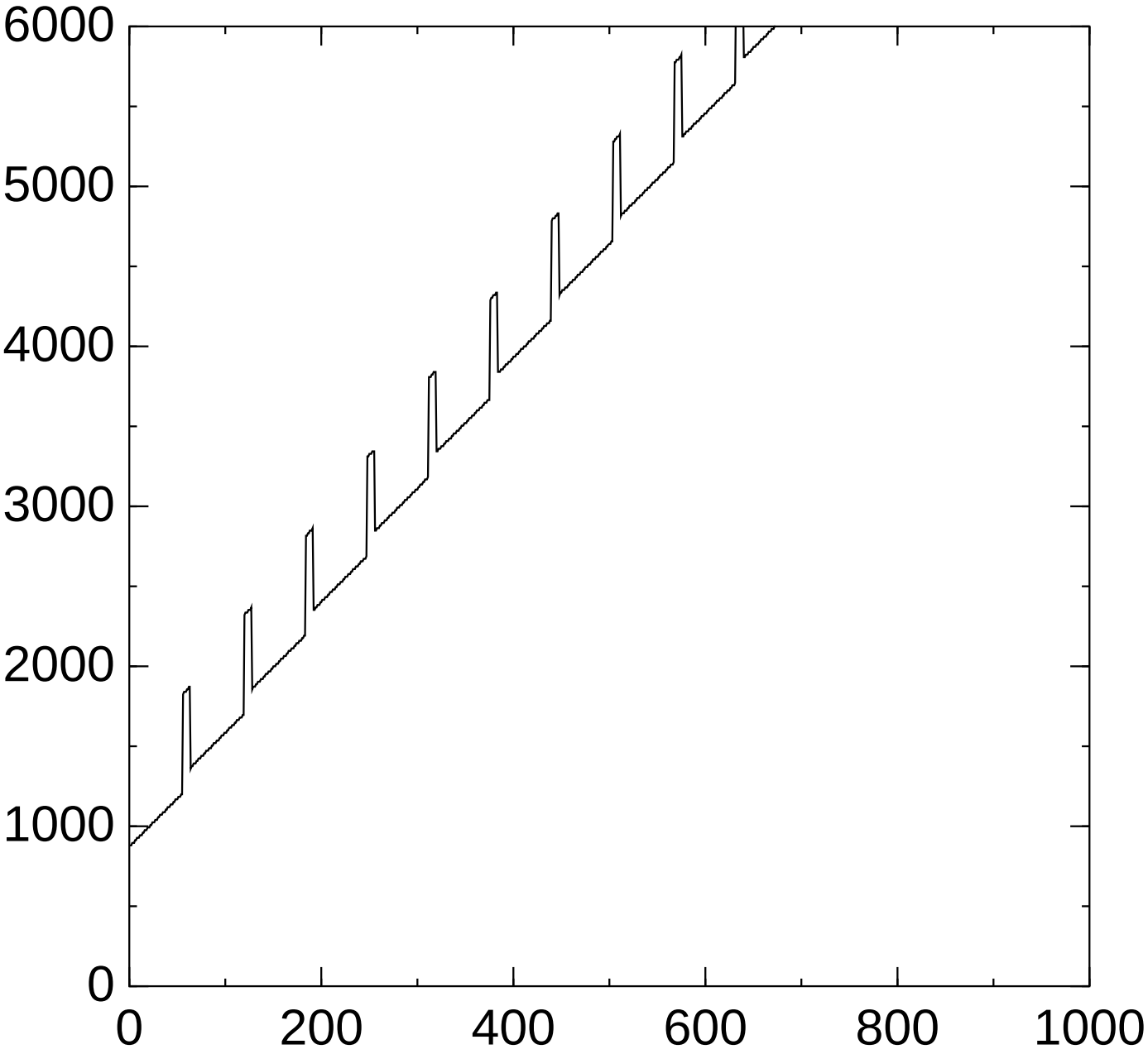
MD5 cycles on `nmi-0104`,
x86 arch, Pentium D f64:



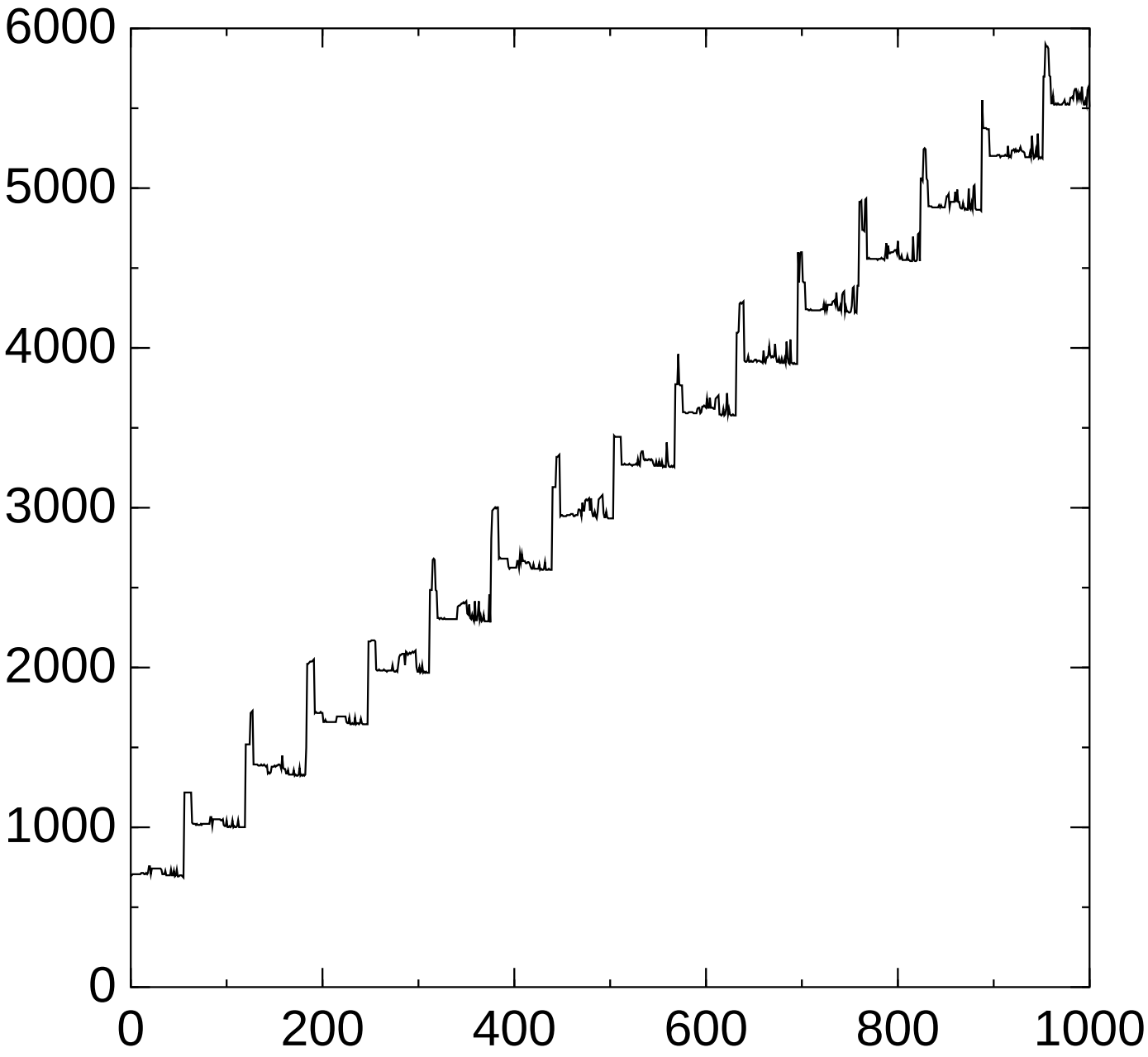
MD5 cycles on `nmi-0020`, ia64 arch, Itanium II:



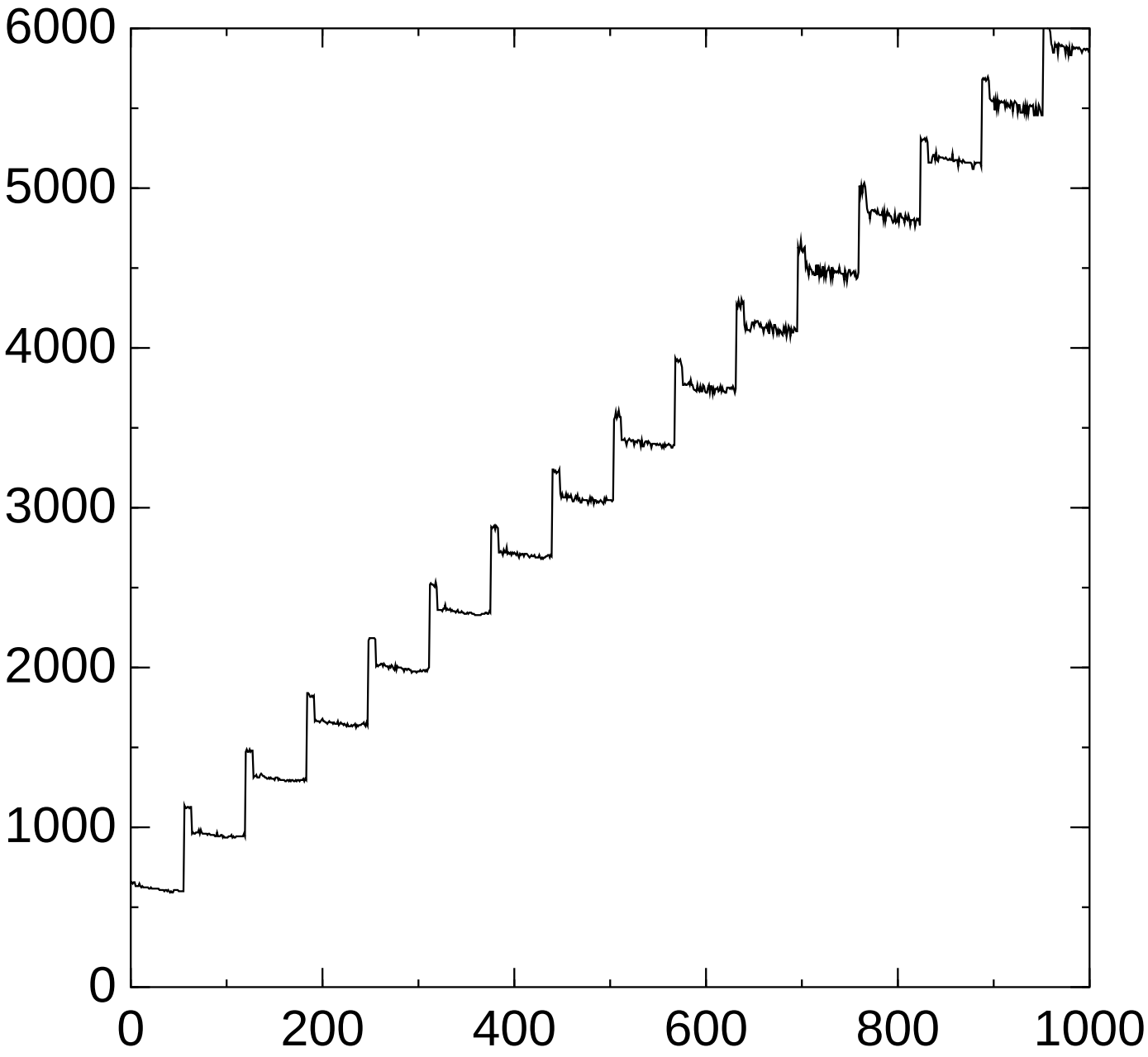
MD5 cycles on gggg,
ppc32 arch, PowerPC G4 7410:



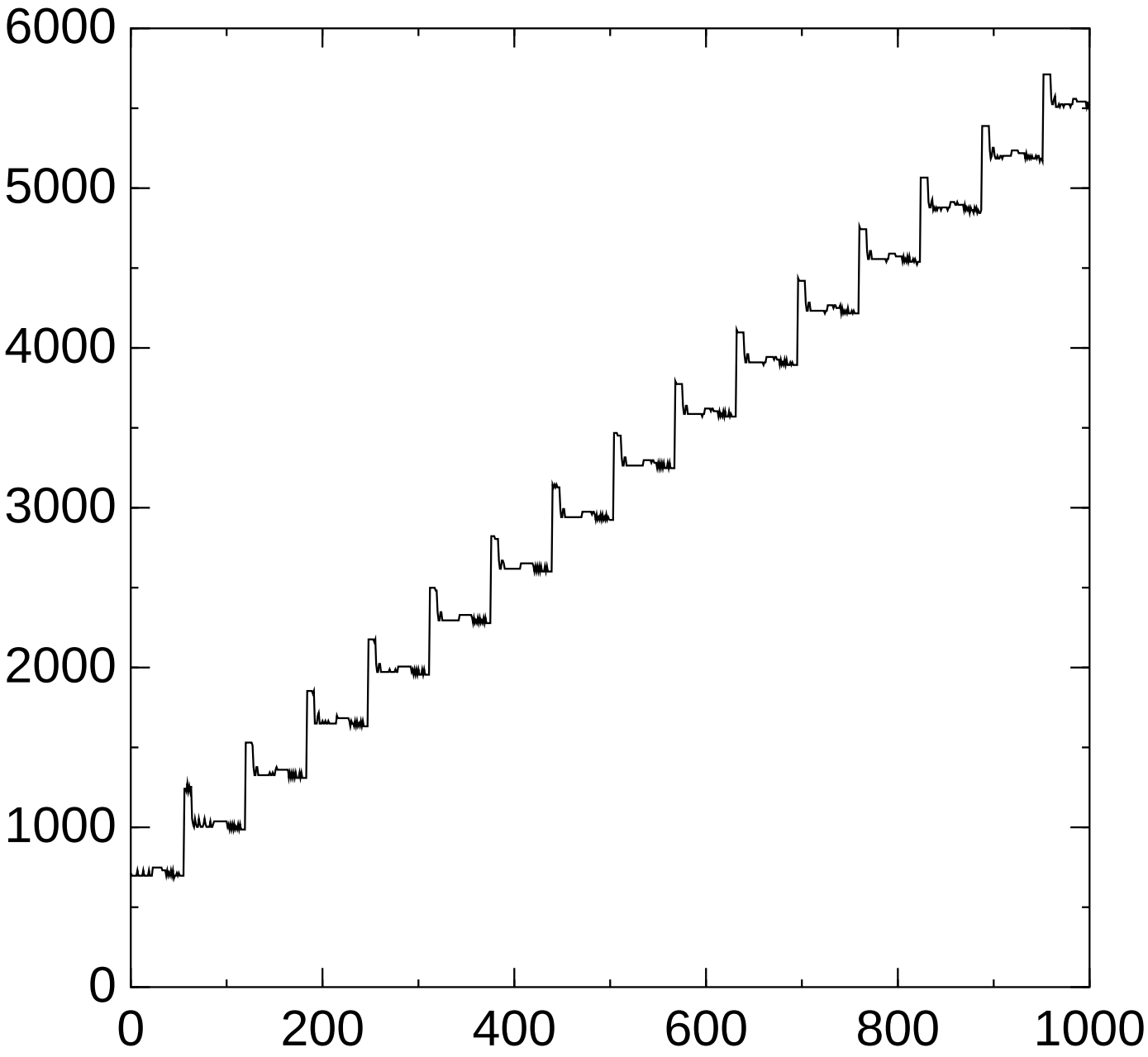
MD5 cycles on `nmi-0056`, amd64 arch, Xeon f41:



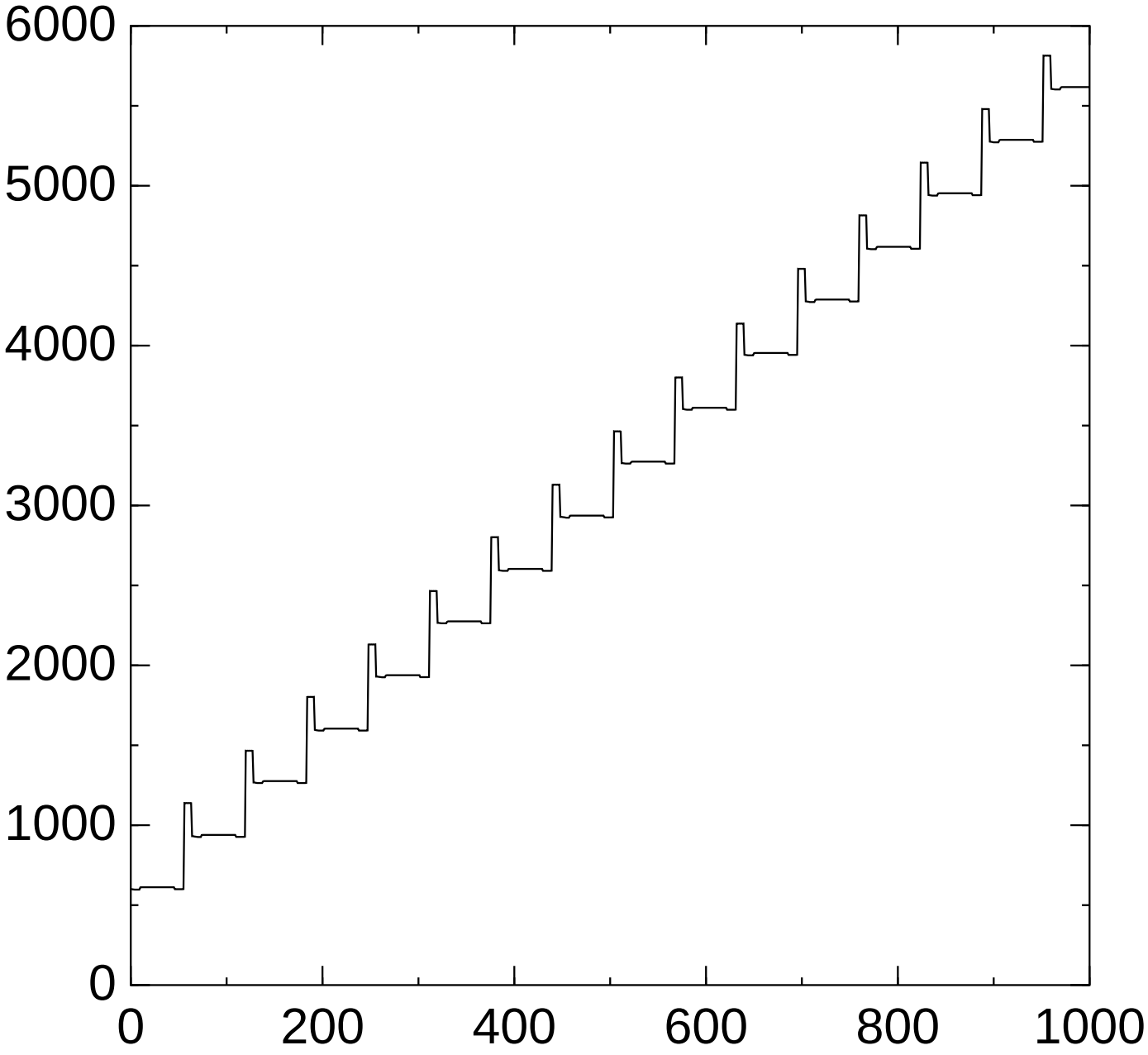
MD5 cycles on `katana`, amd64 arch, Core 2 Duo:



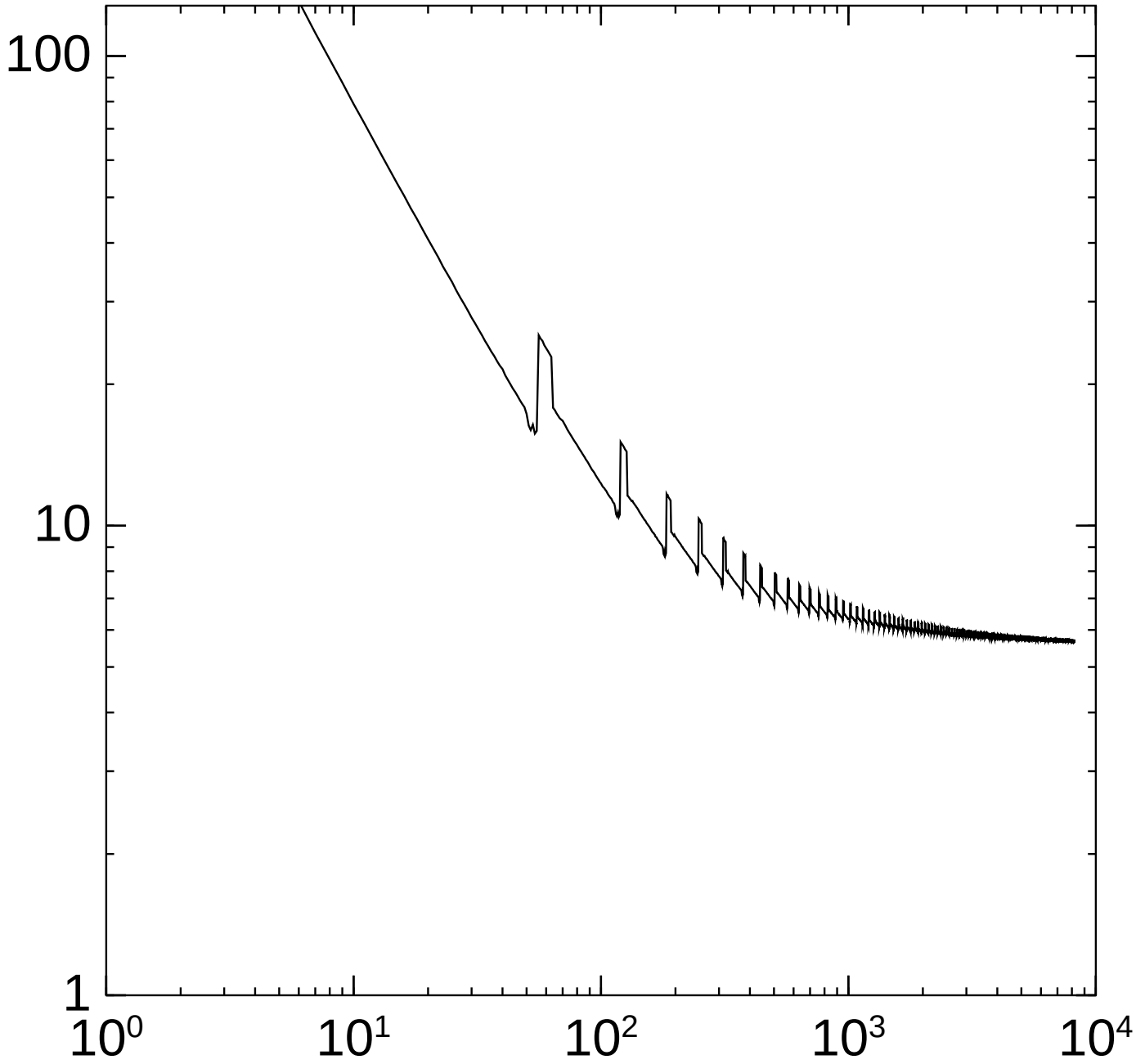
MD5 cycles on `nmi-0104`, amd64 arch, Pentium D f64:



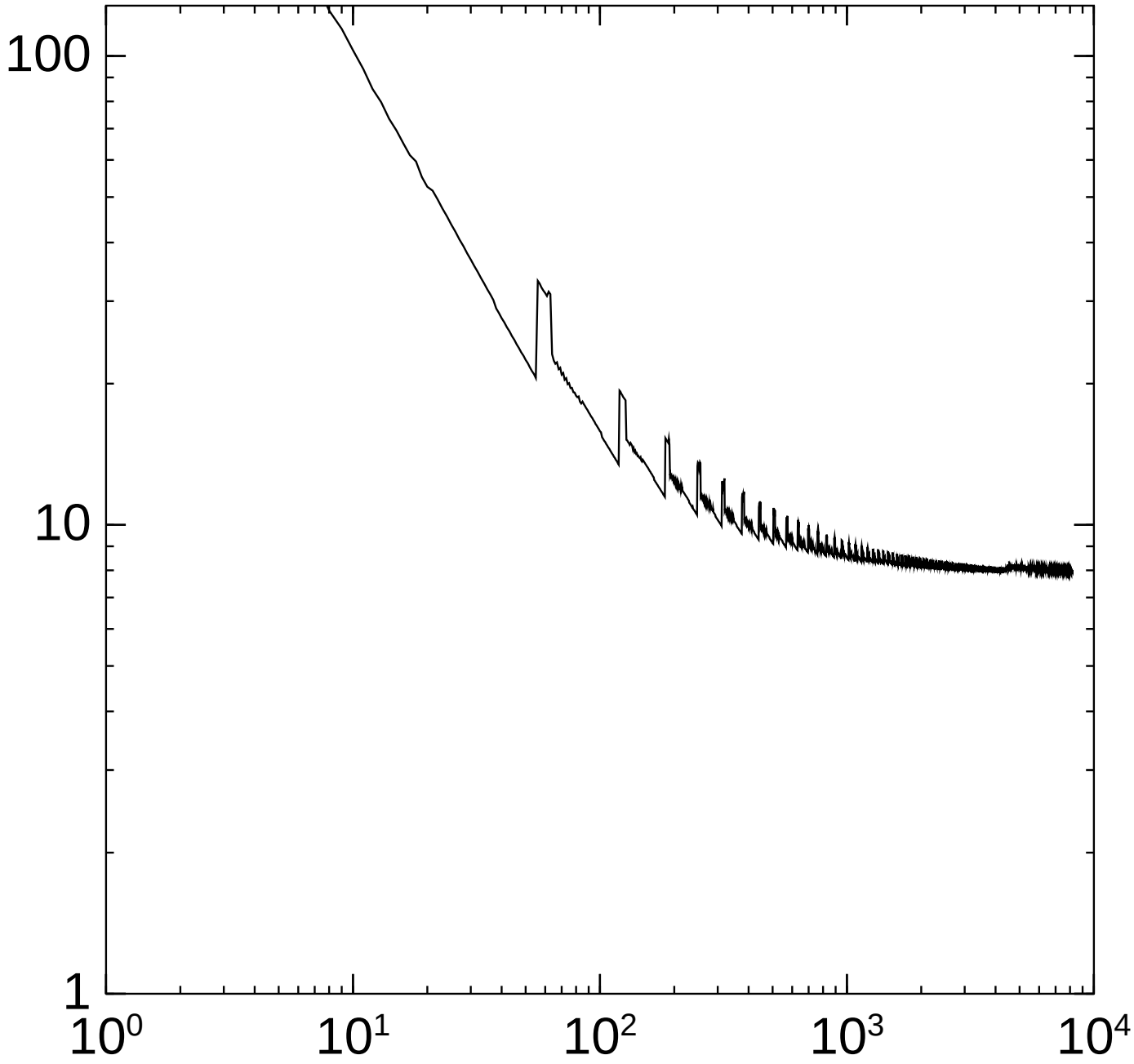
MD5 cycles on `mace`,
amd64 arch, Athlon 64 X2:



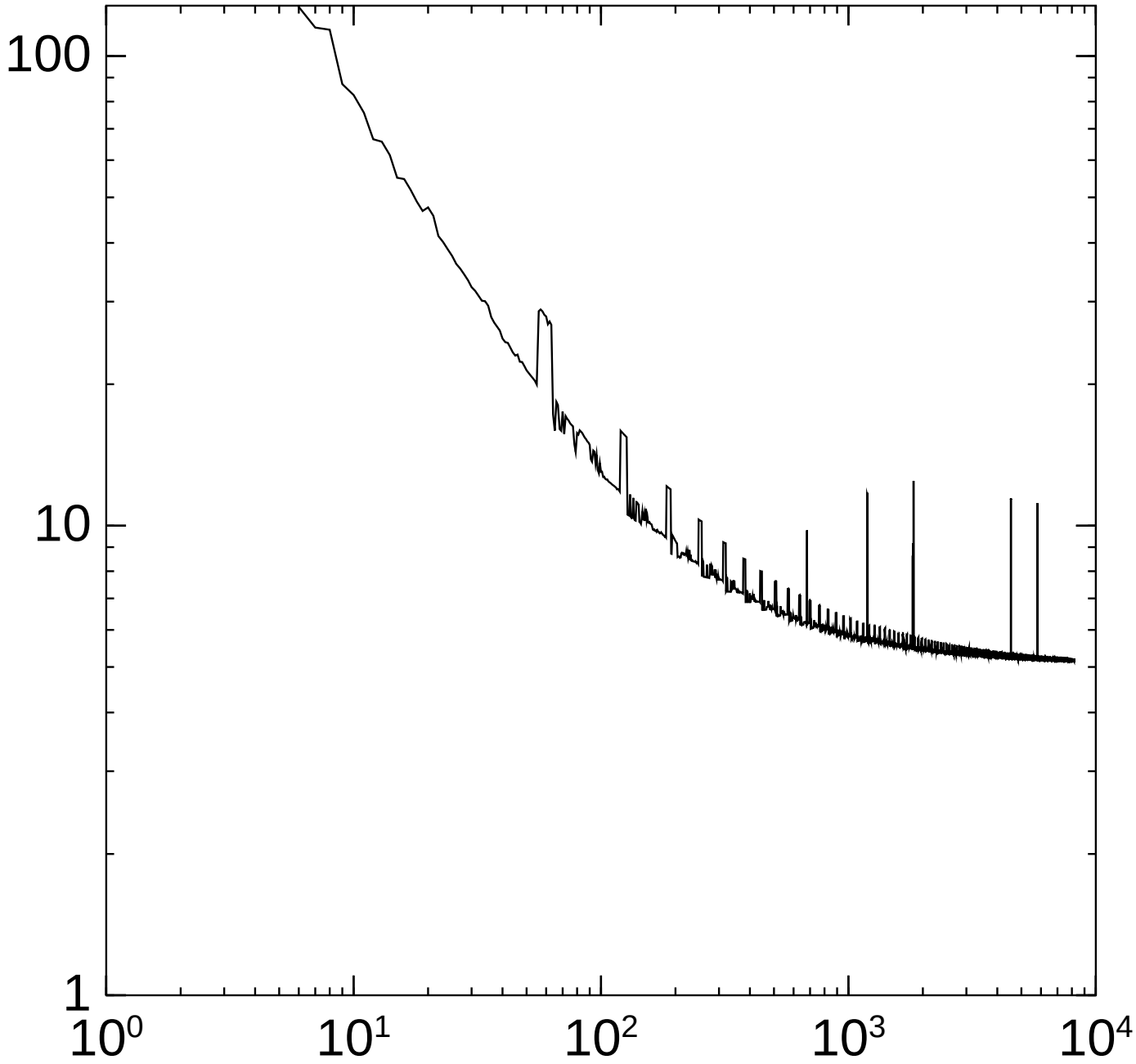
MD5 cycles/byte on `orpheus`, x86 architecture, Pentium 3 672:



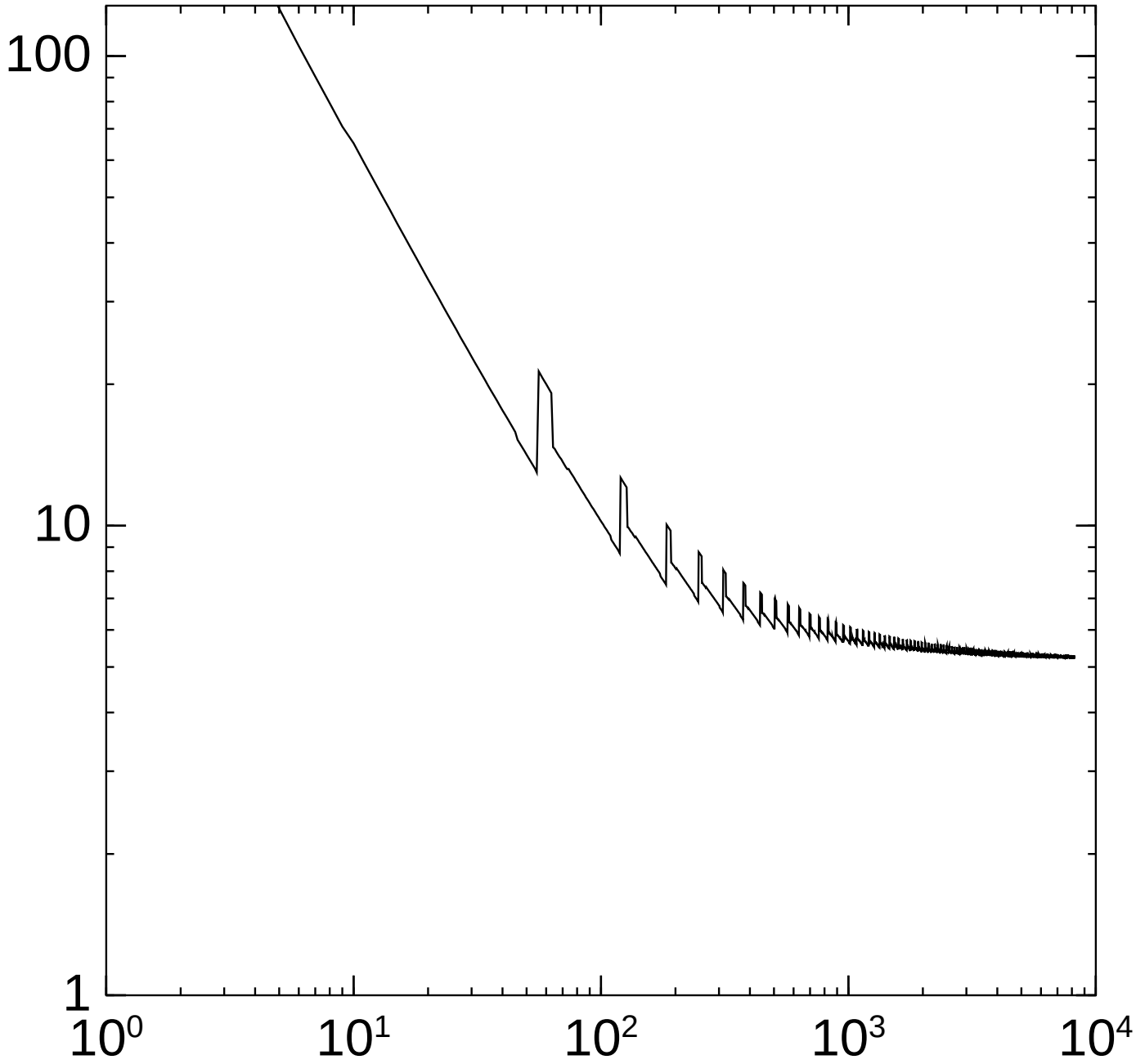
MD5 cycles/byte on `fireball`, x86 arch, Pentium 4 f12:



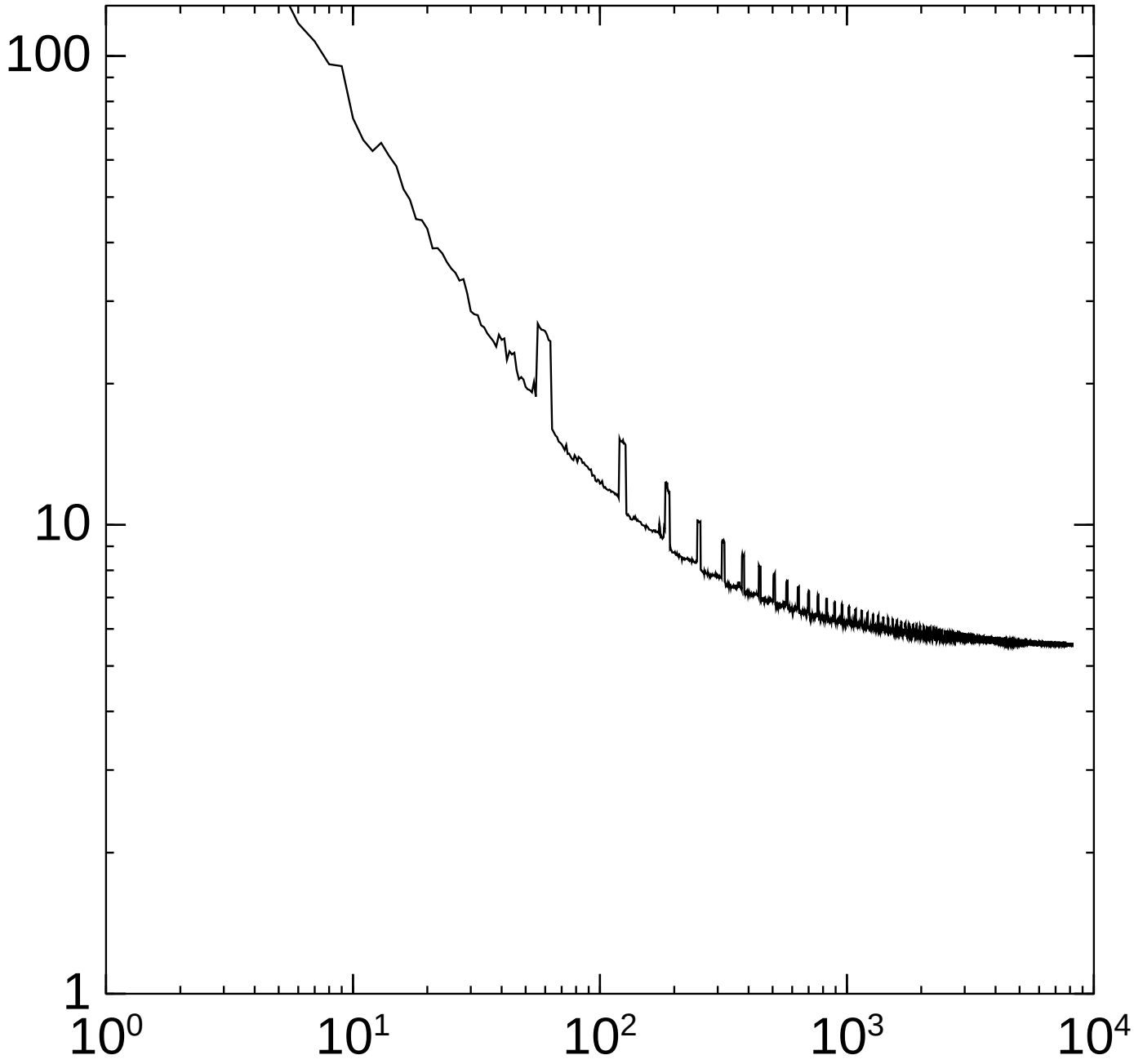
MD5 cycles/byte on `nmi-0056`, x86 arch, Xeon f41:



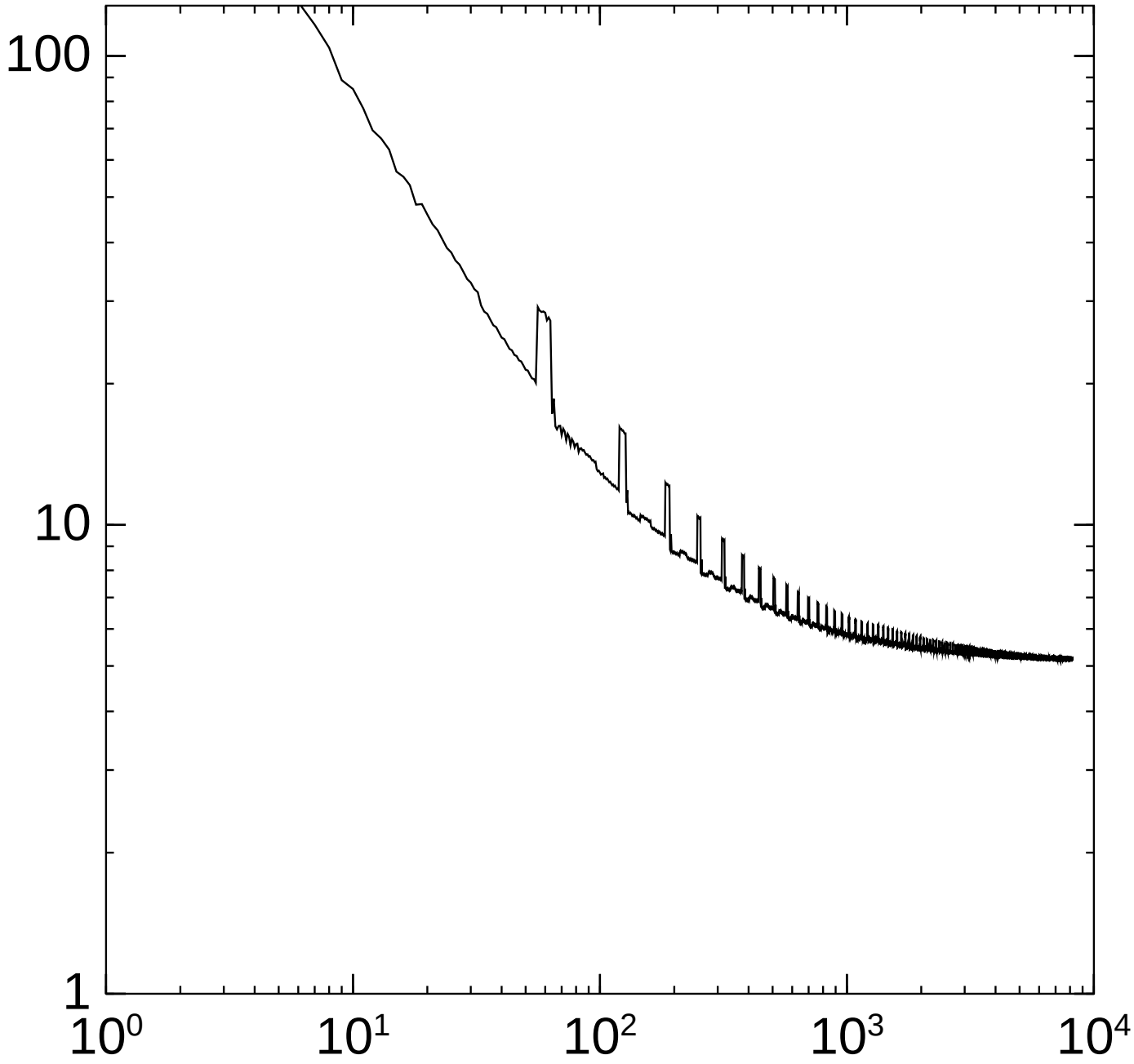
MD5 cycles/byte on `thoth`, x86 arch, Athlon:



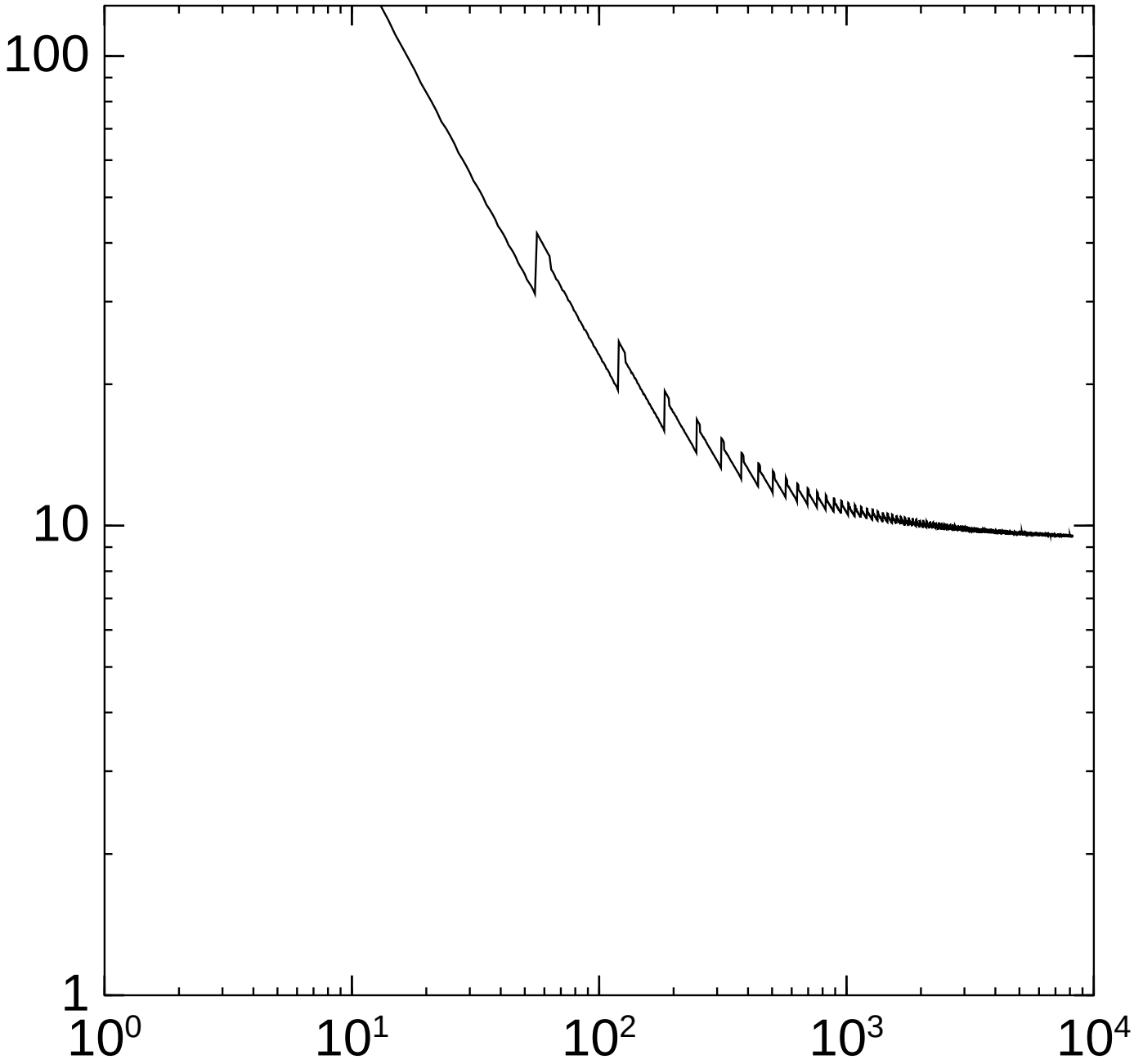
MD5 cycles/byte on `katana`,
x86 arch, Core 2 Duo:



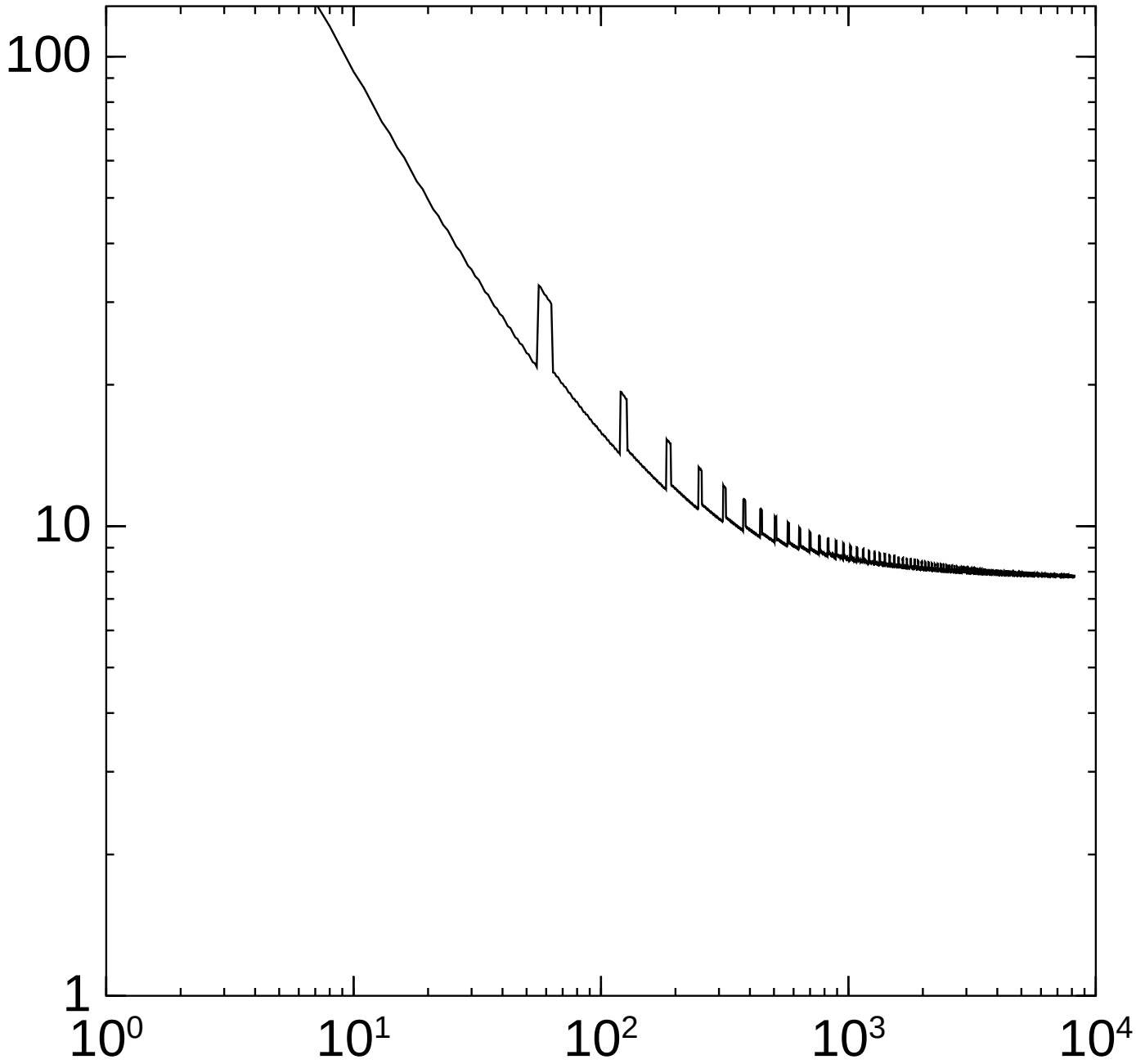
MD5 cycles/byte on `nmi-0104`, x86 arch, Pentium D f64:



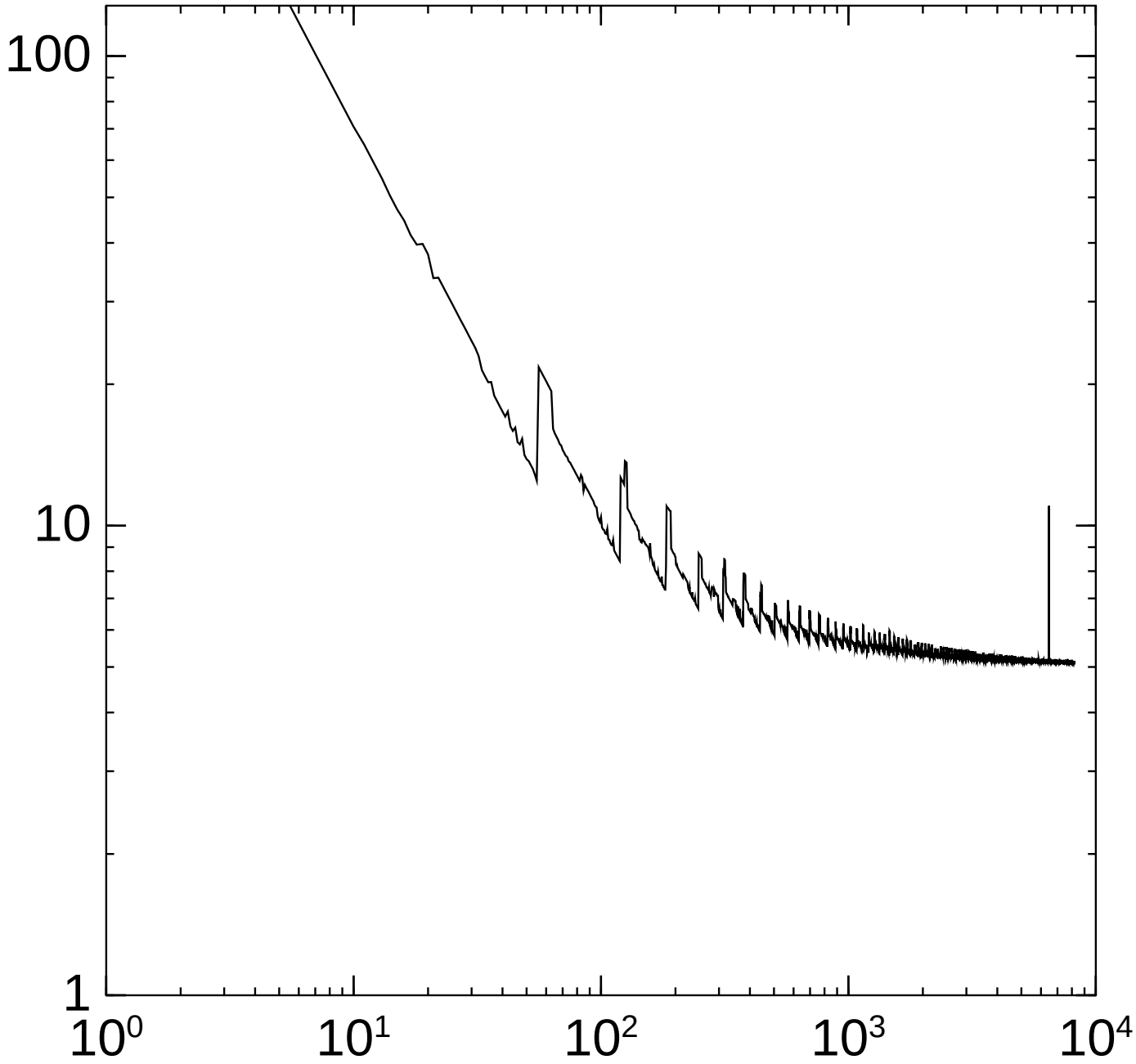
MD5 cycles/byte on `nmi-0020`, ia64 arch, Itanium II:



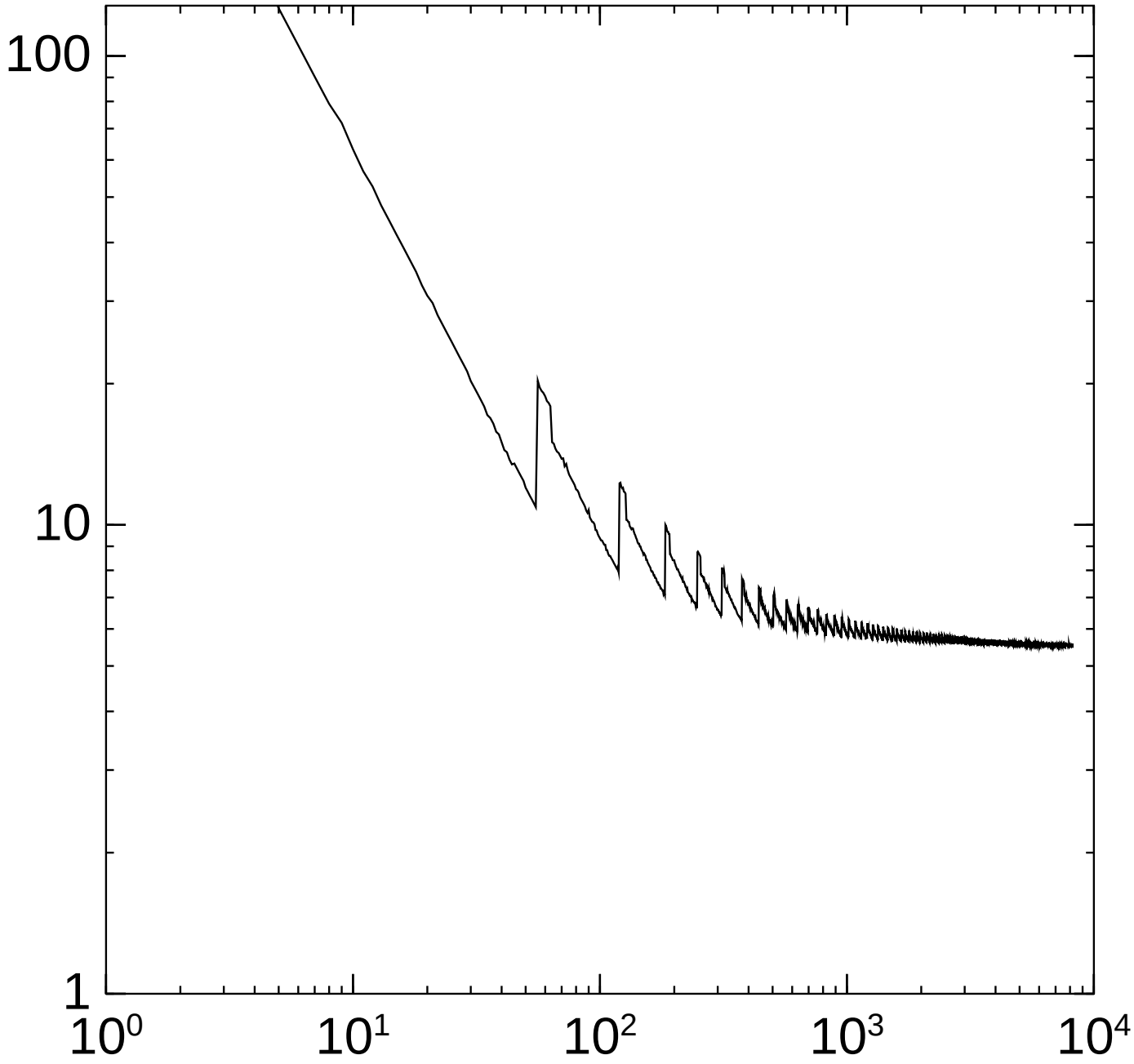
MD5 cycles/byte on gggg,
ppc32 arch, PowerPC G4 7410:



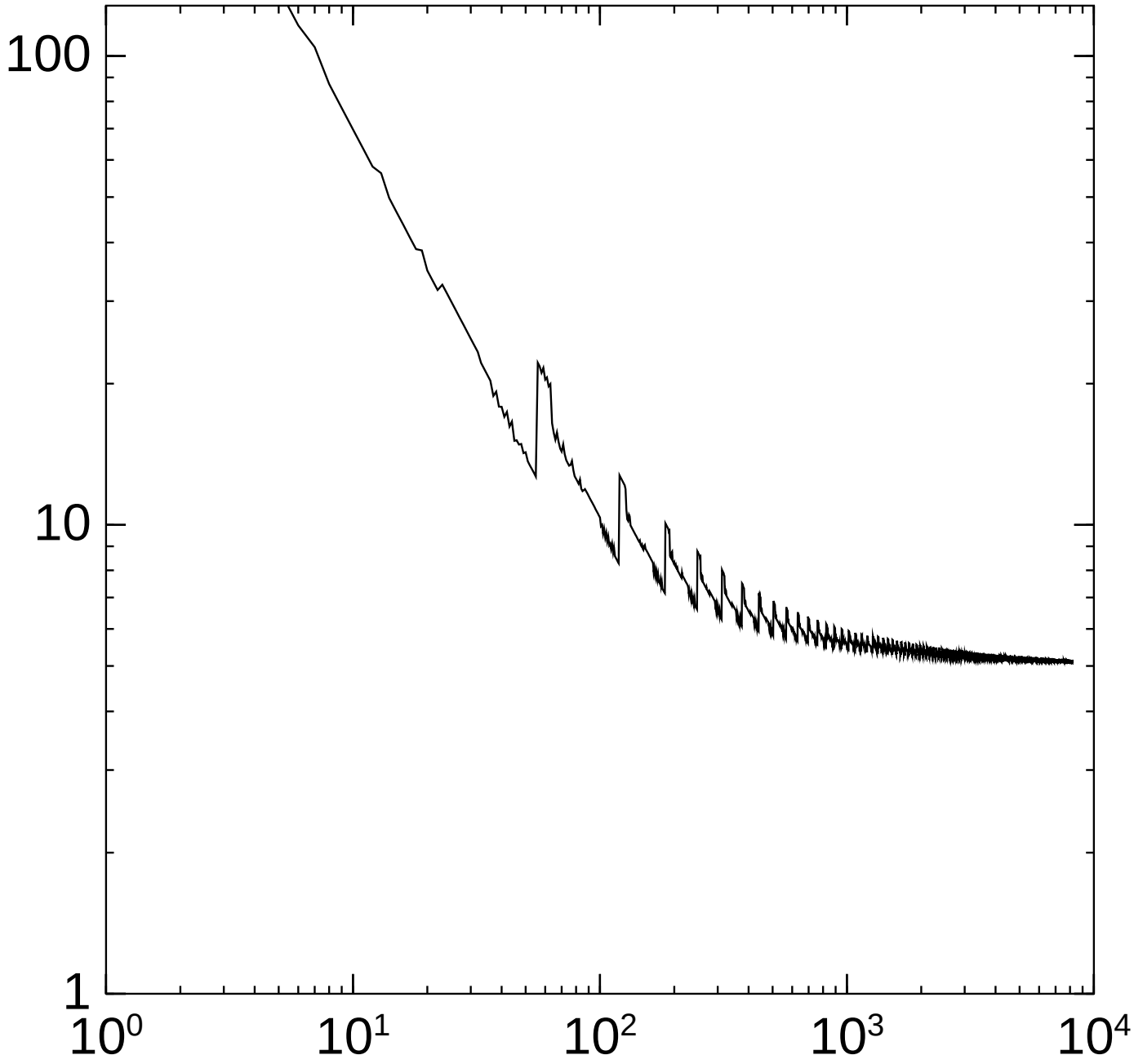
MD5 cycles/byte on `nmi-0056`, amd64 arch, Xeon f41:



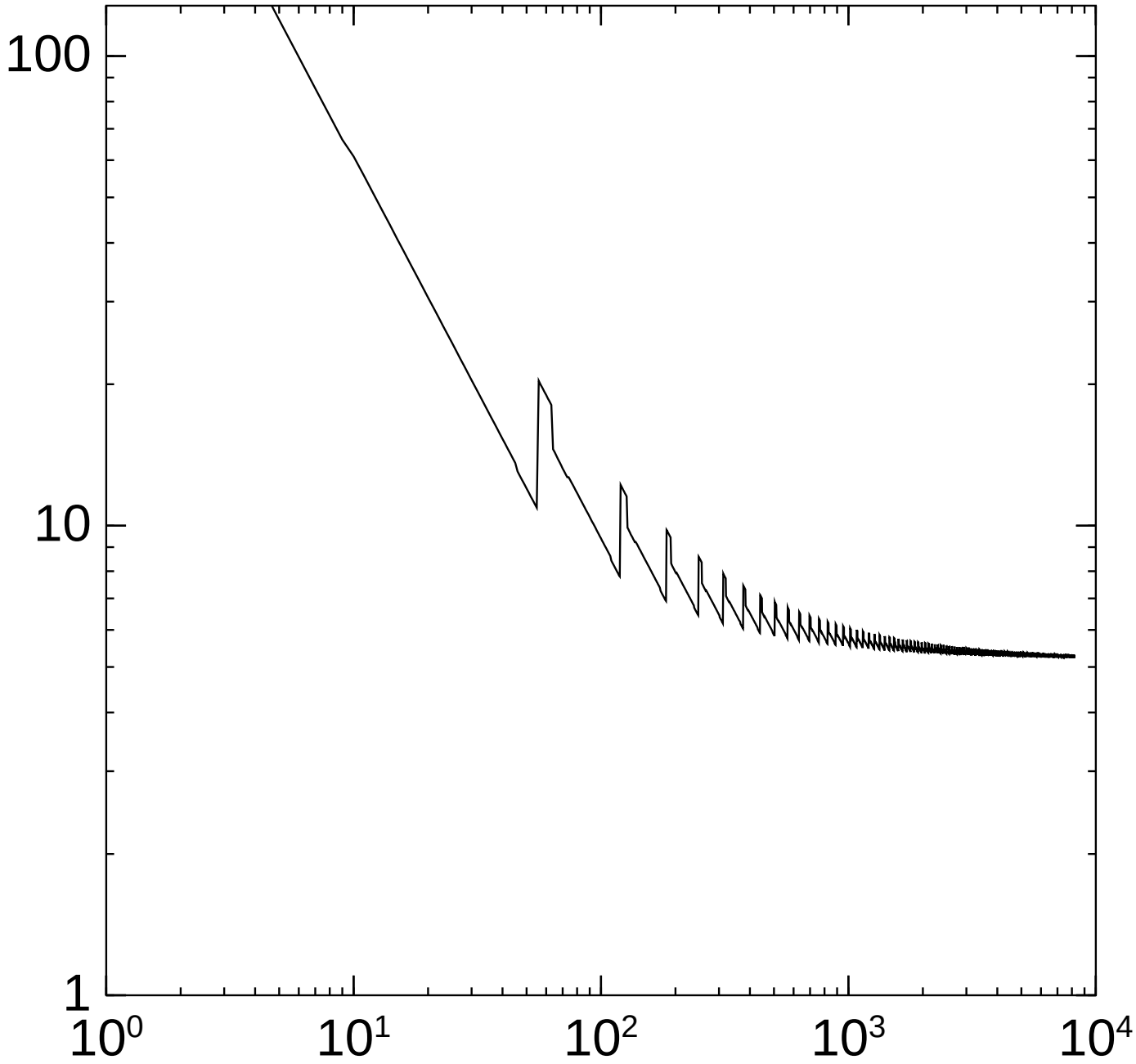
MD5 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



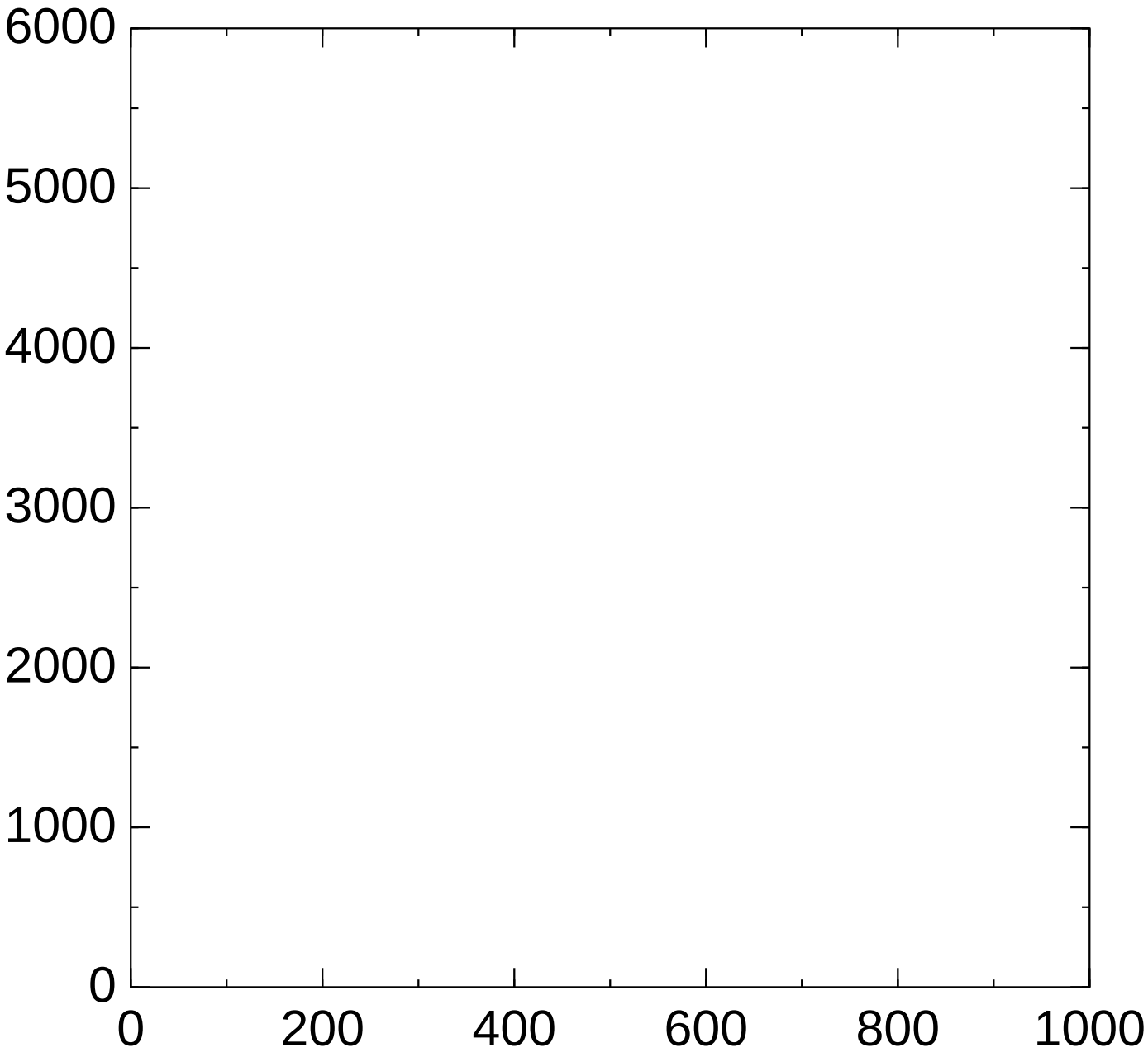
MD5 cycles/byte on `nmi-0104`, amd64 arch, Pentium D f64:



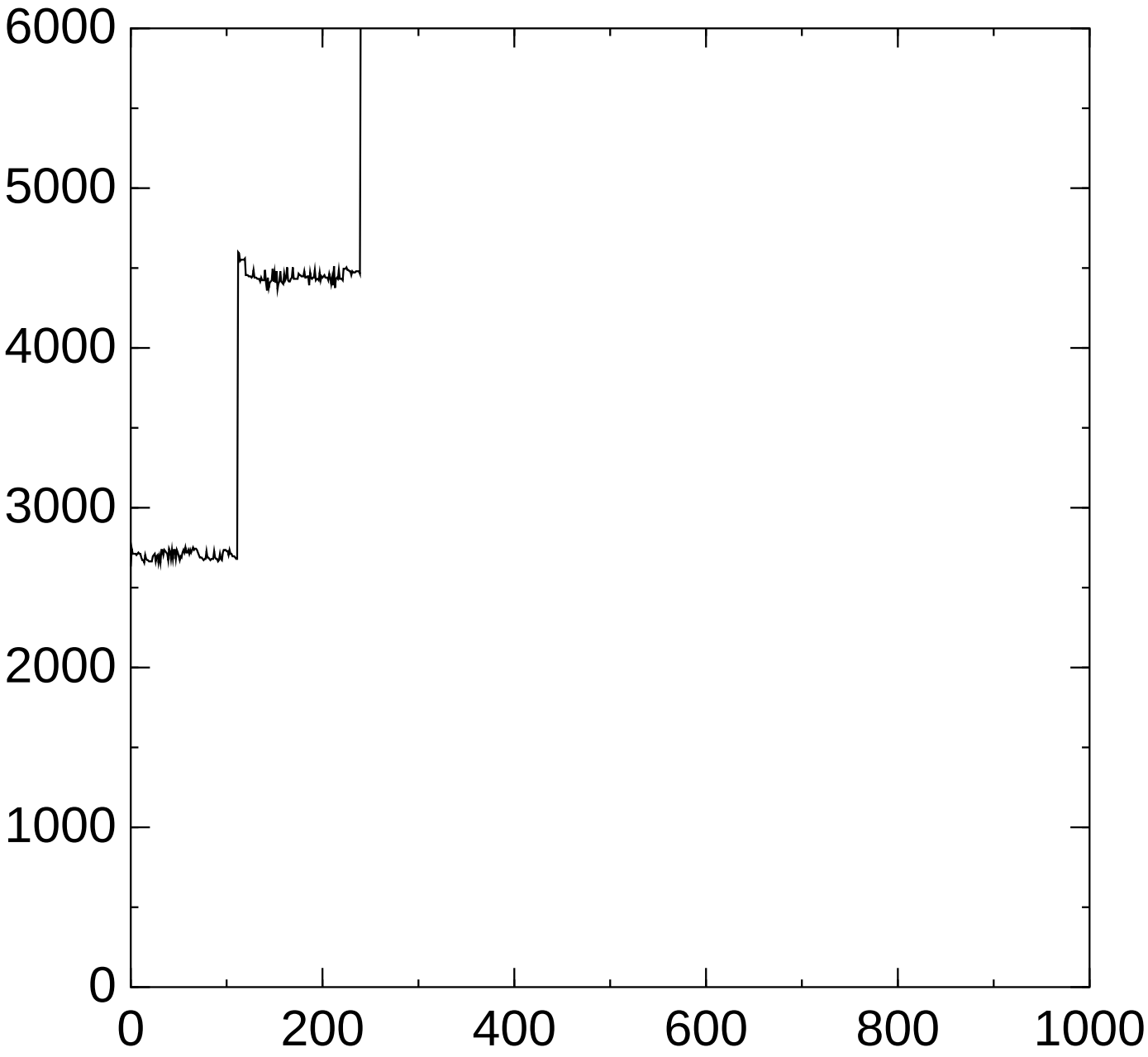
MD5 cycles/byte on `mace`, amd64 arch, Athlon 64 X2:



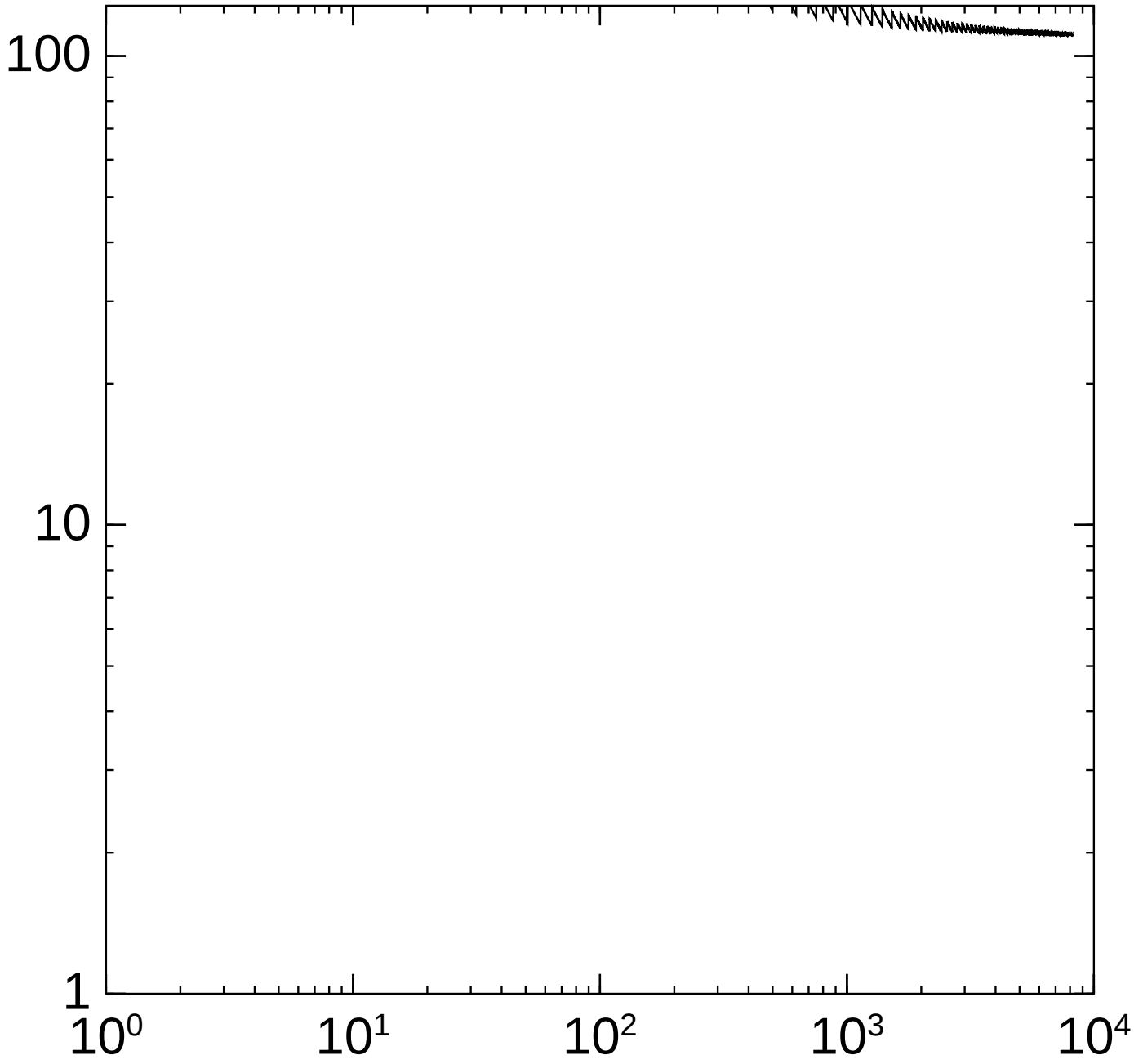
SHA-512 cycles on `orpheus`,
x86 arch, Pentium 3 672:



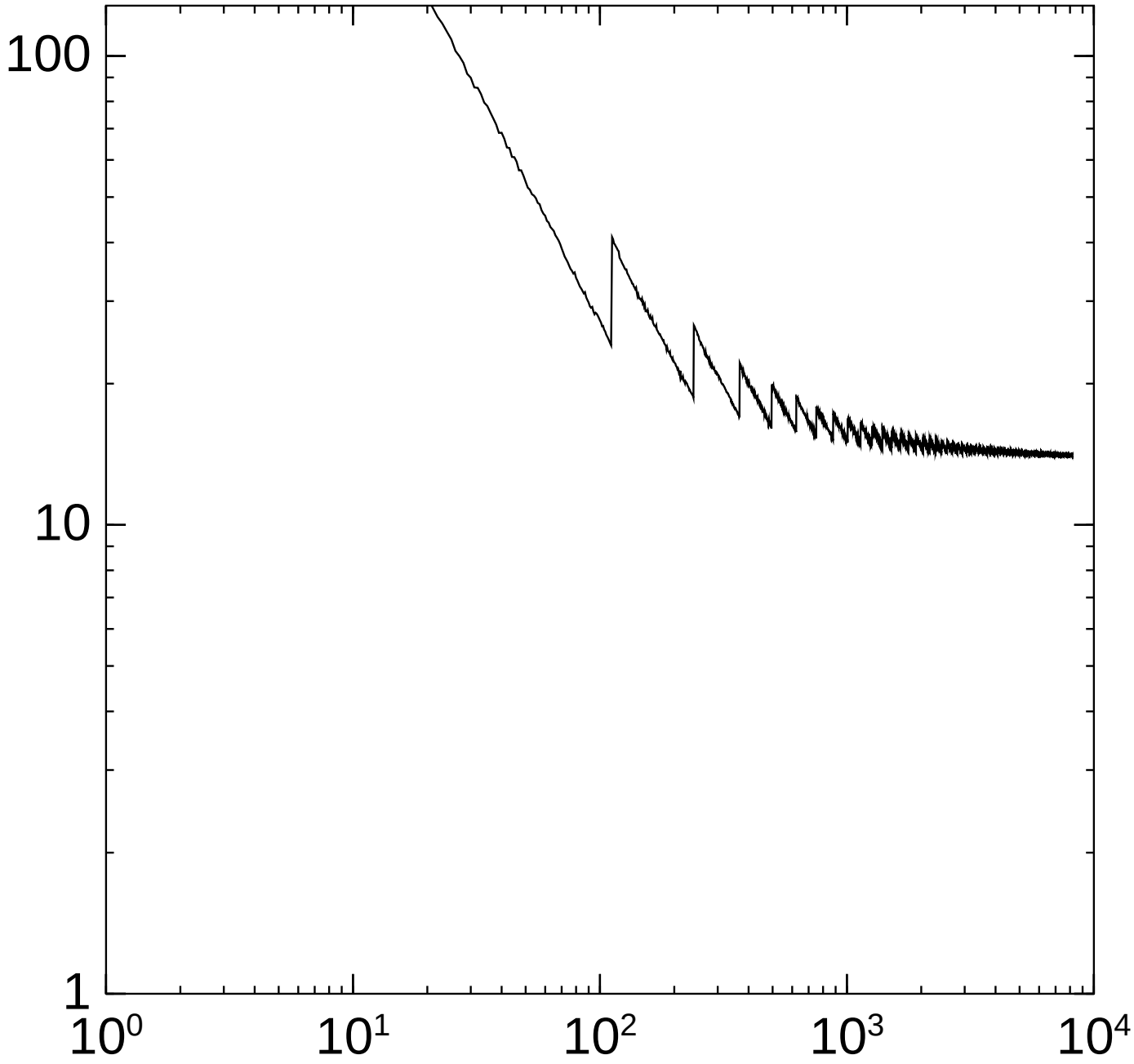
SHA-512 cycles on `katana`, amd64 arch, Core 2 Duo:



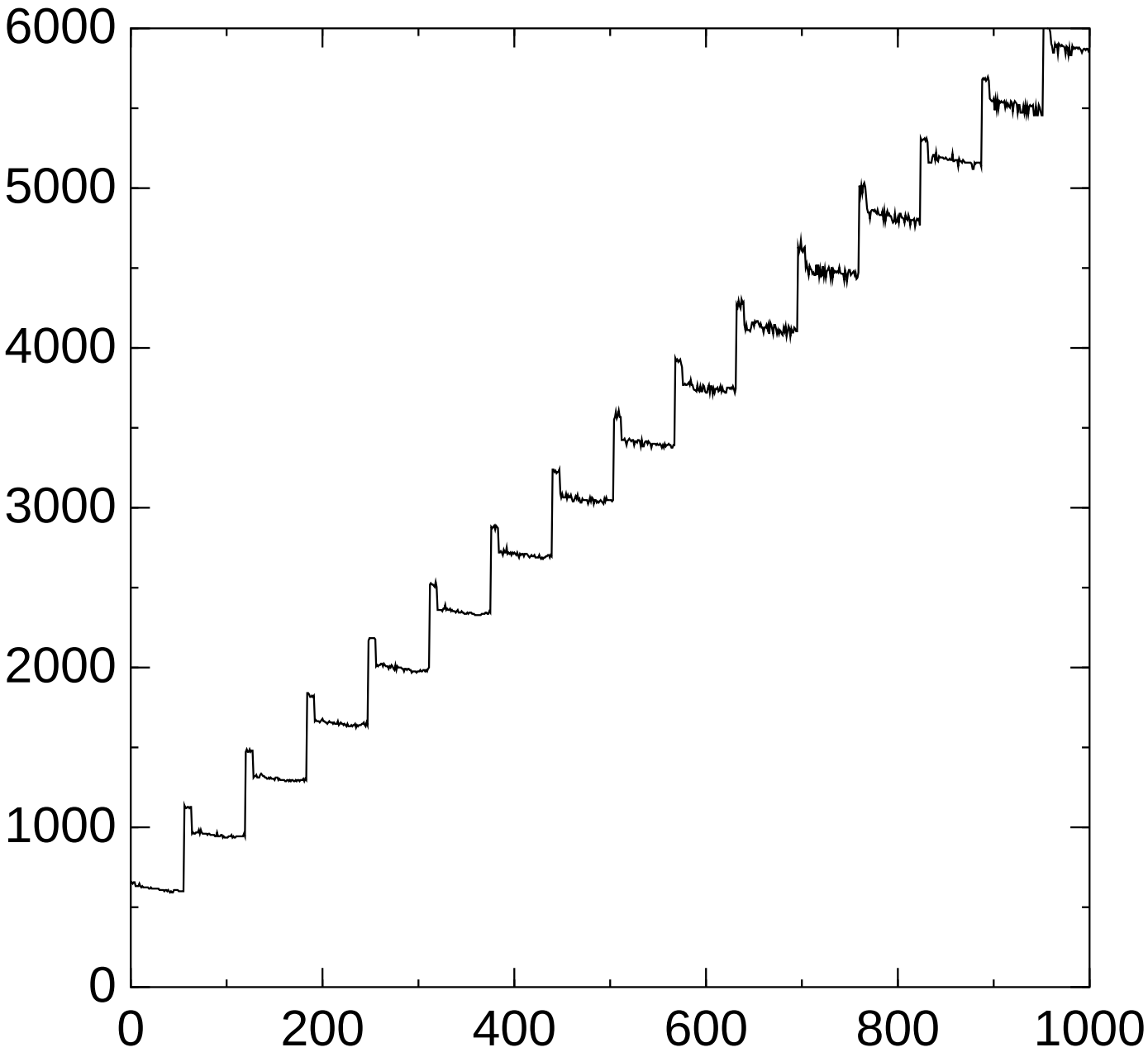
SHA-512 cycles/byte on `orpheus`, x86 arch, Pentium 3 672:



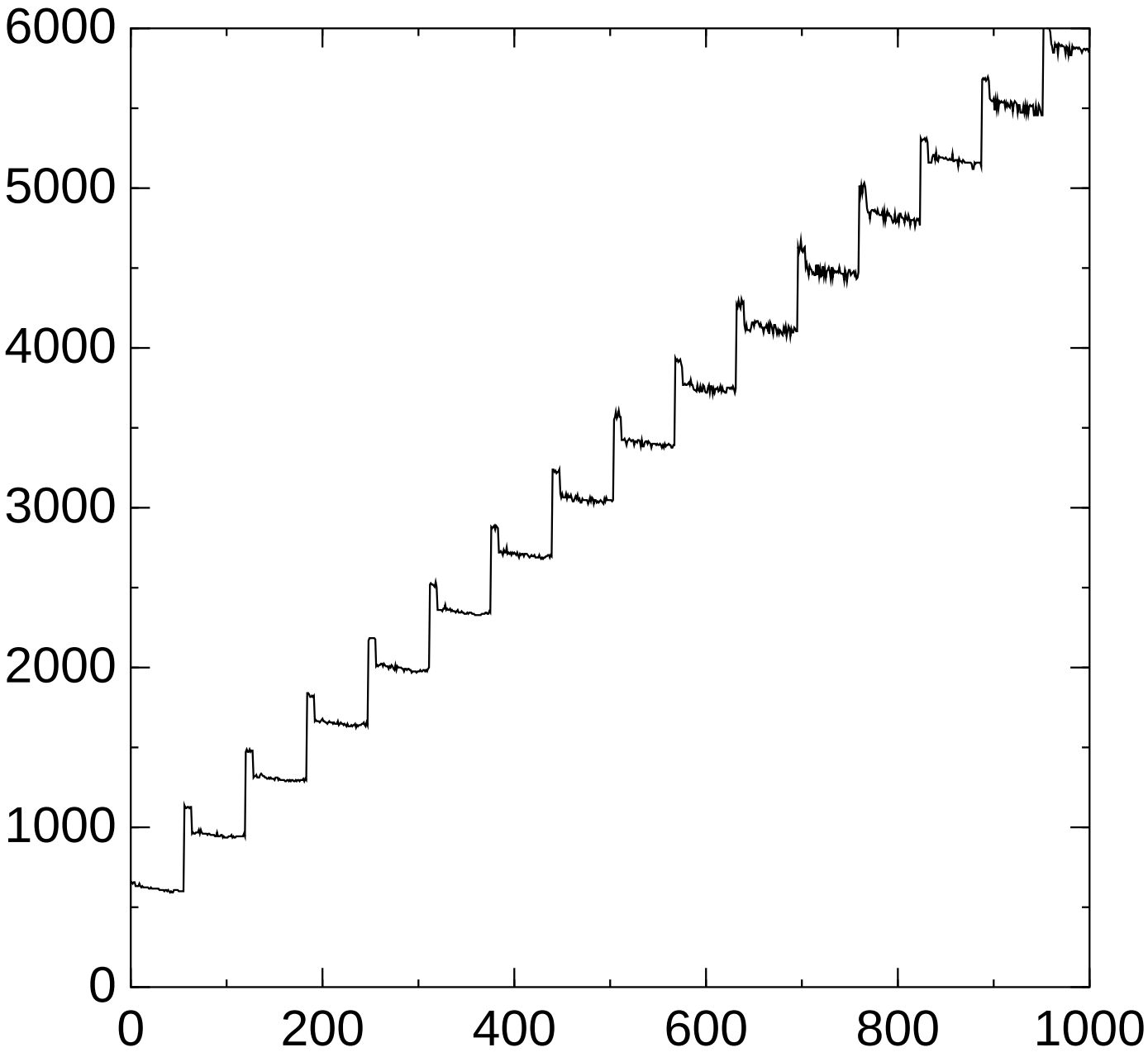
SHA-512 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



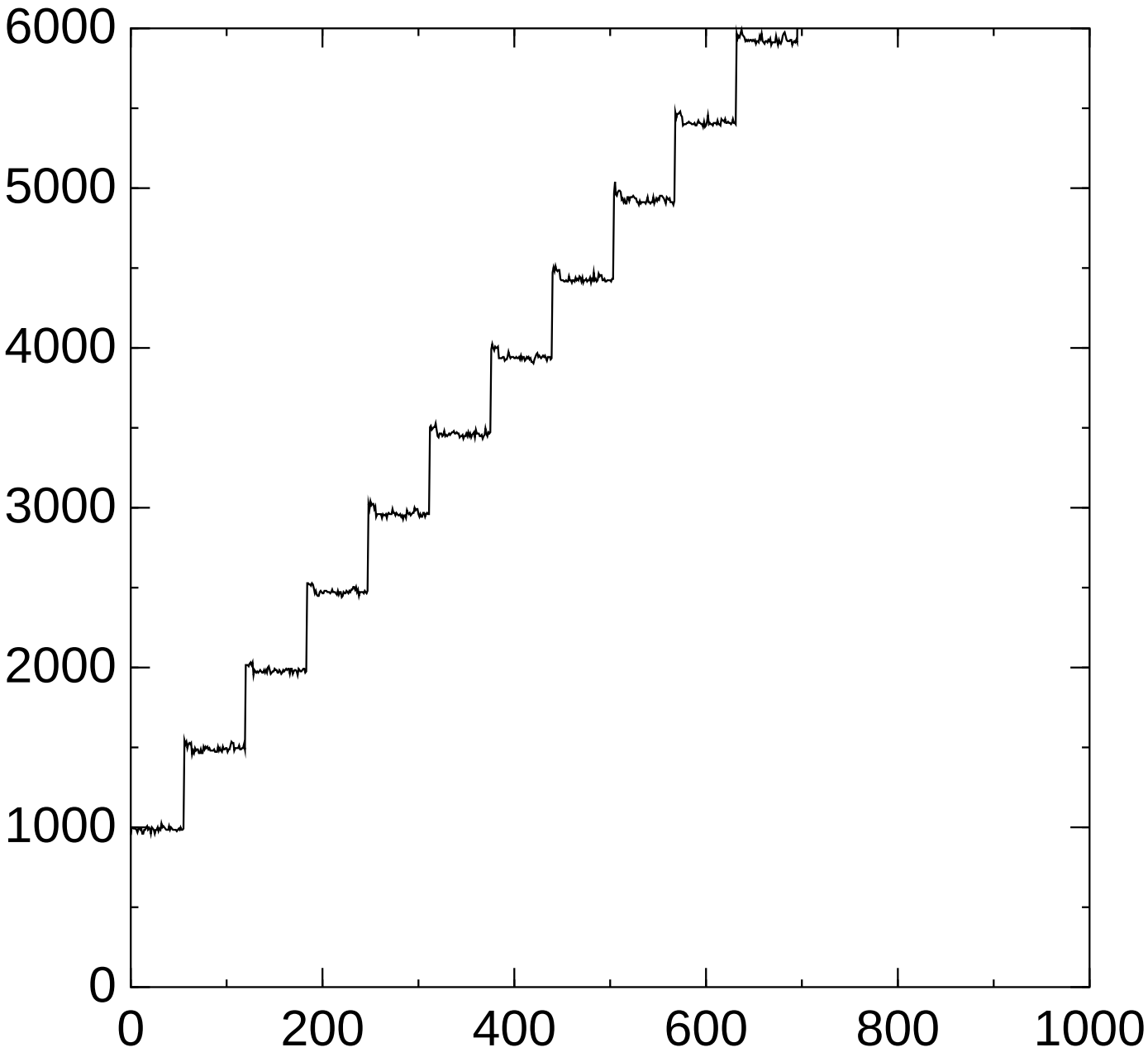
MD4 cycles on `katana`, amd64 arch, Core 2 Duo:



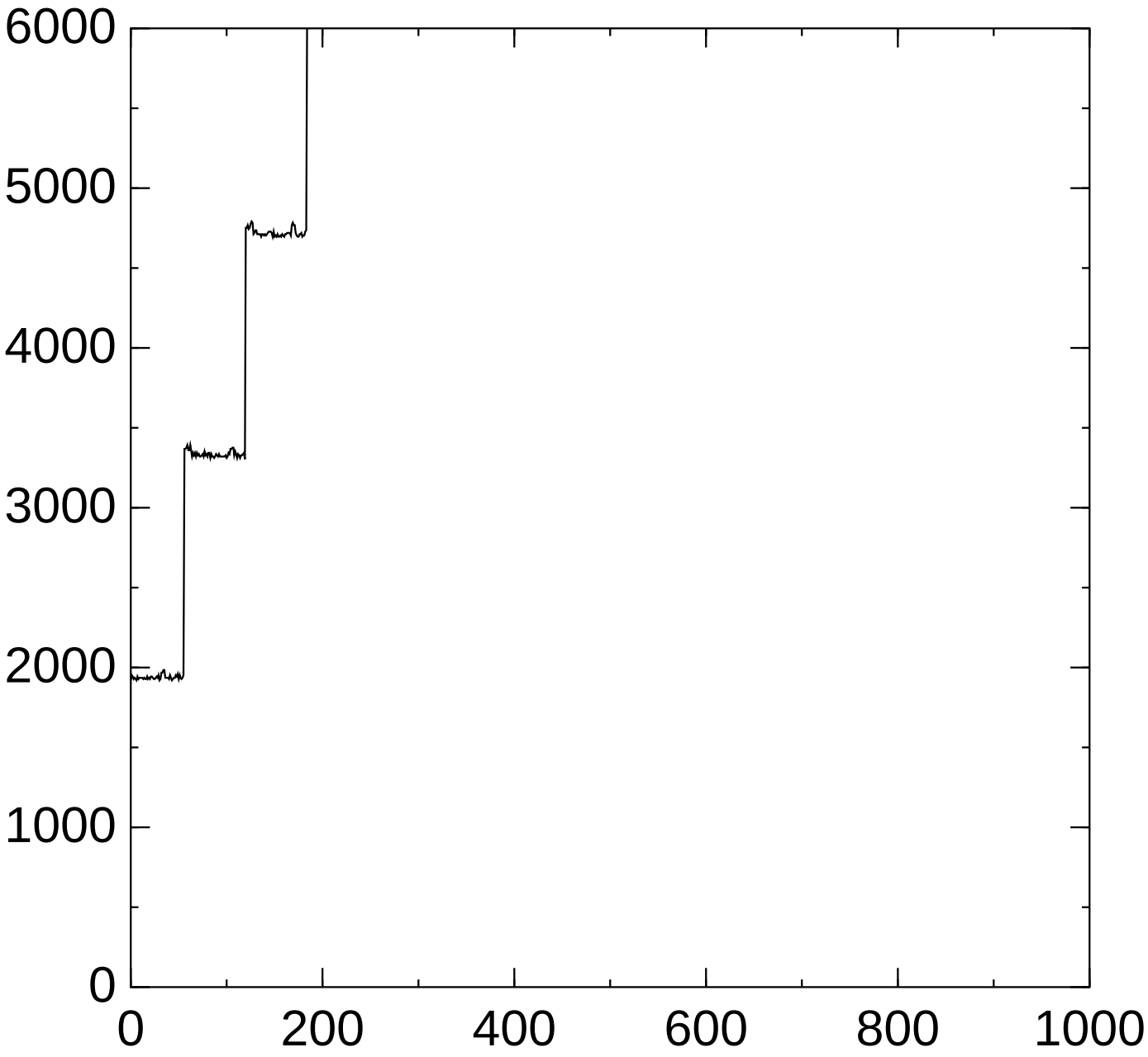
MD5 cycles on `katana`, amd64 arch, Core 2 Duo:



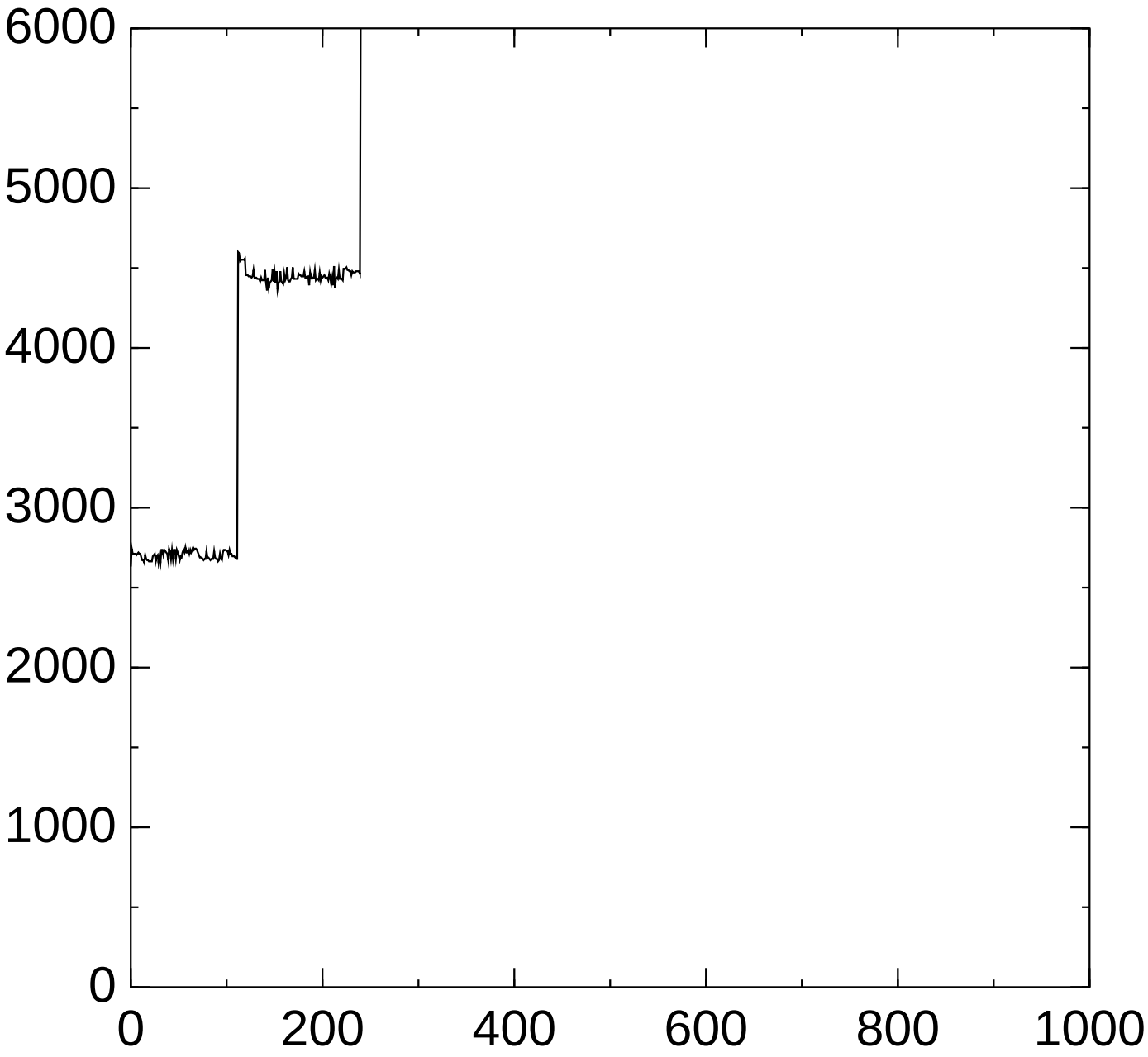
SHA-1 cycles on `katana`, amd64 arch, Core 2 Duo:



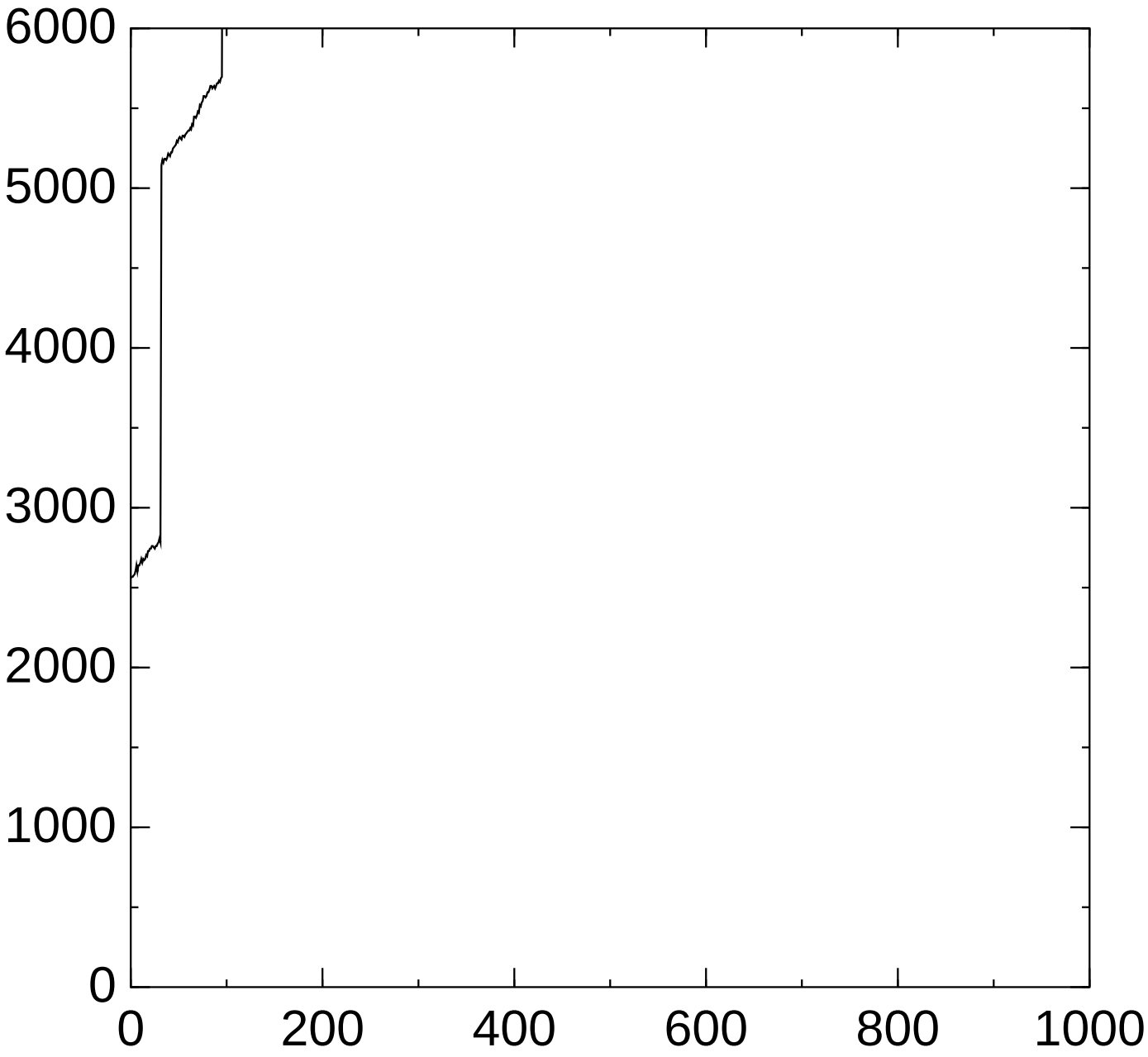
SHA-256 cycles on `katana`, amd64 arch, Core 2 Duo:



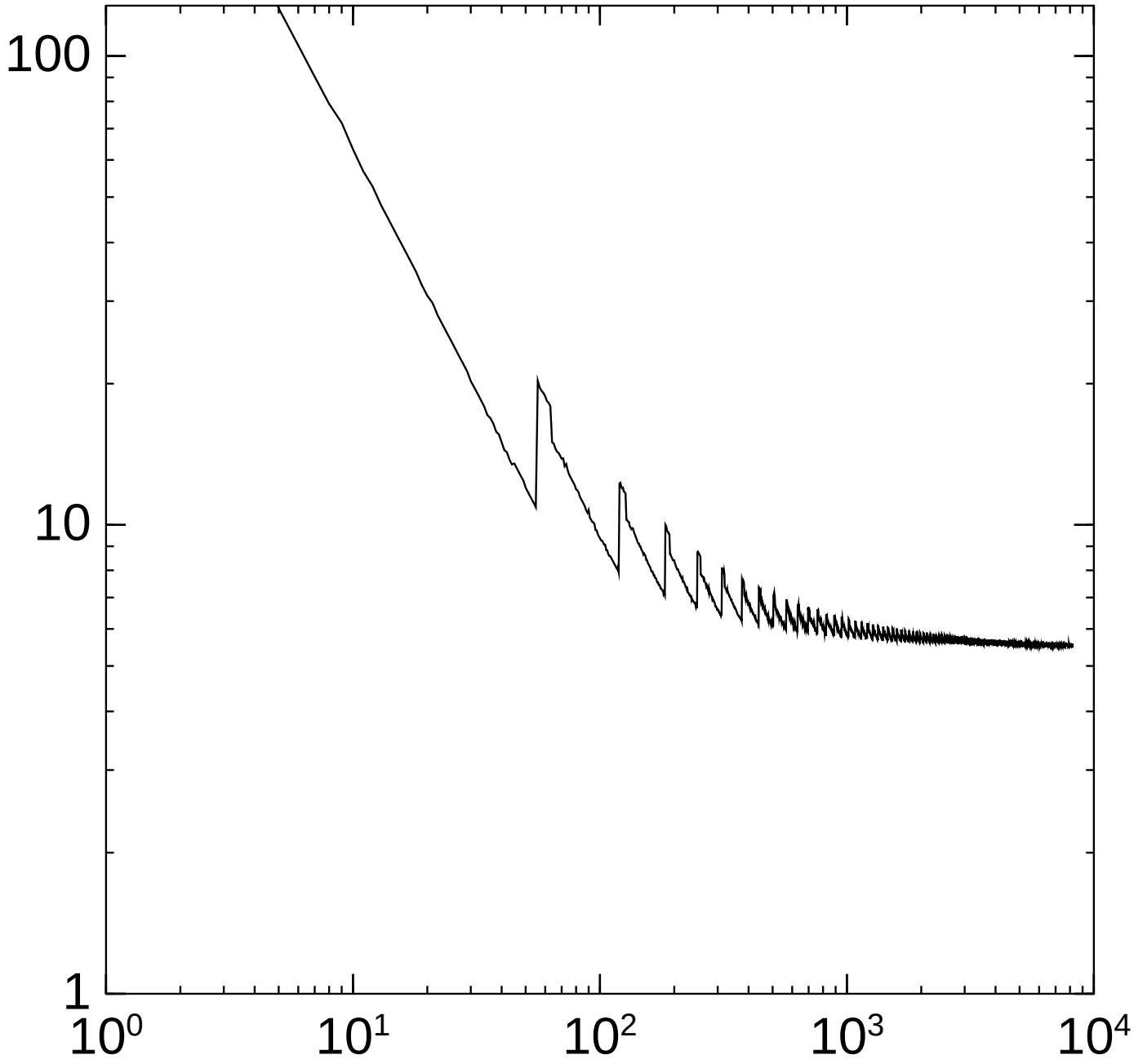
SHA-512 cycles on `katana`, amd64 arch, Core 2 Duo:



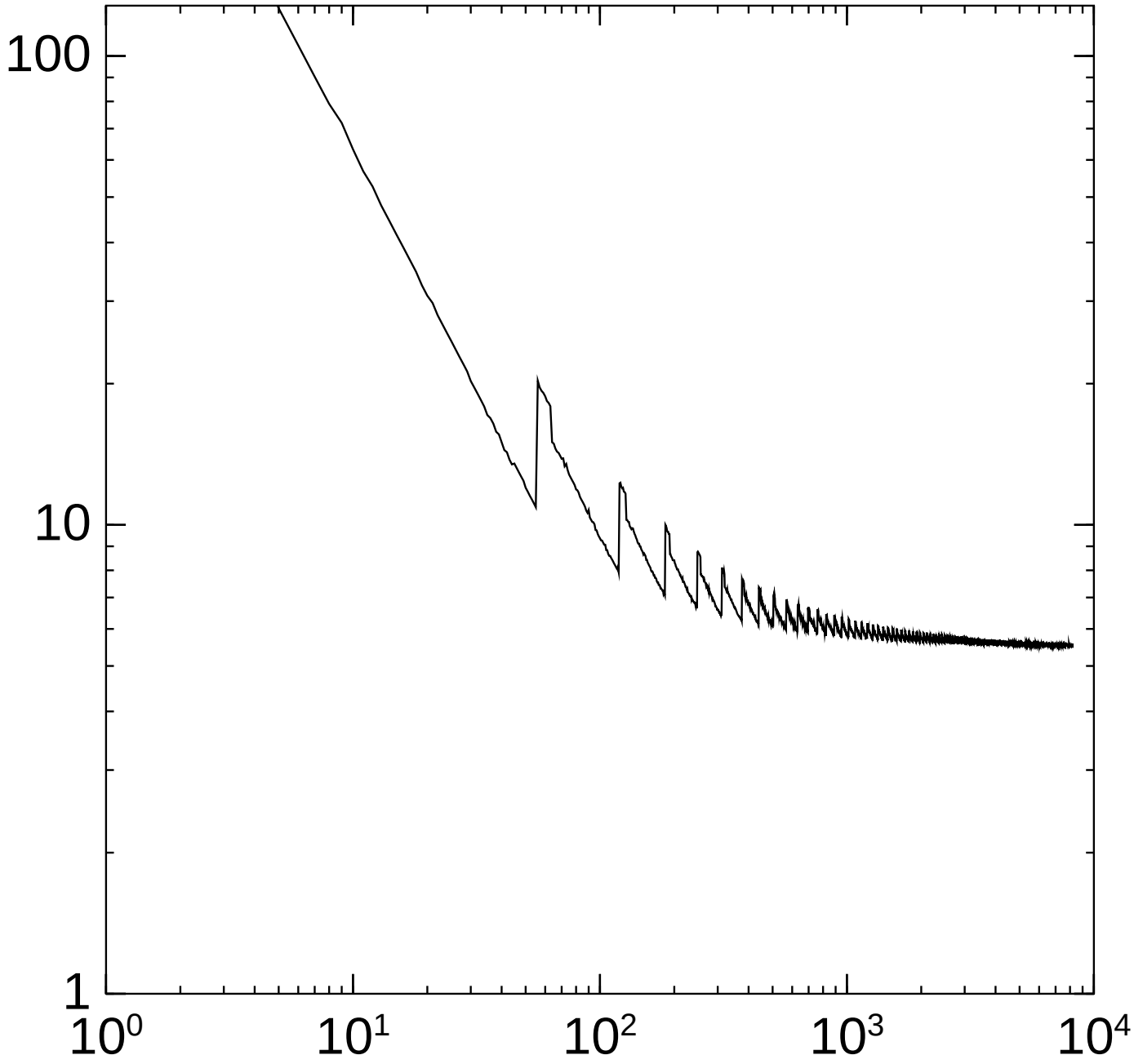
Whirlpool cycles on `katana`, amd64 arch, Core 2 Duo:



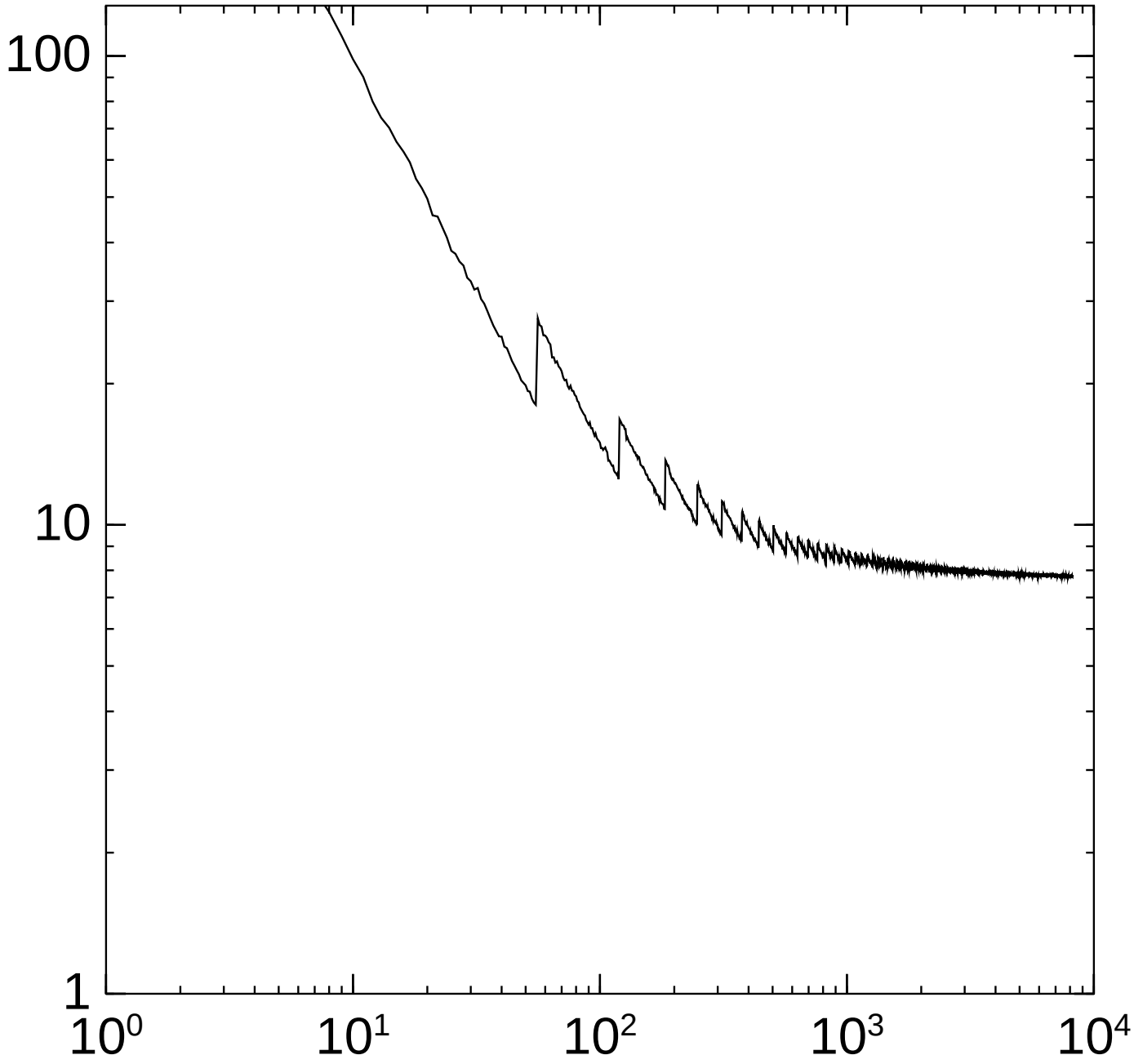
MD4 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



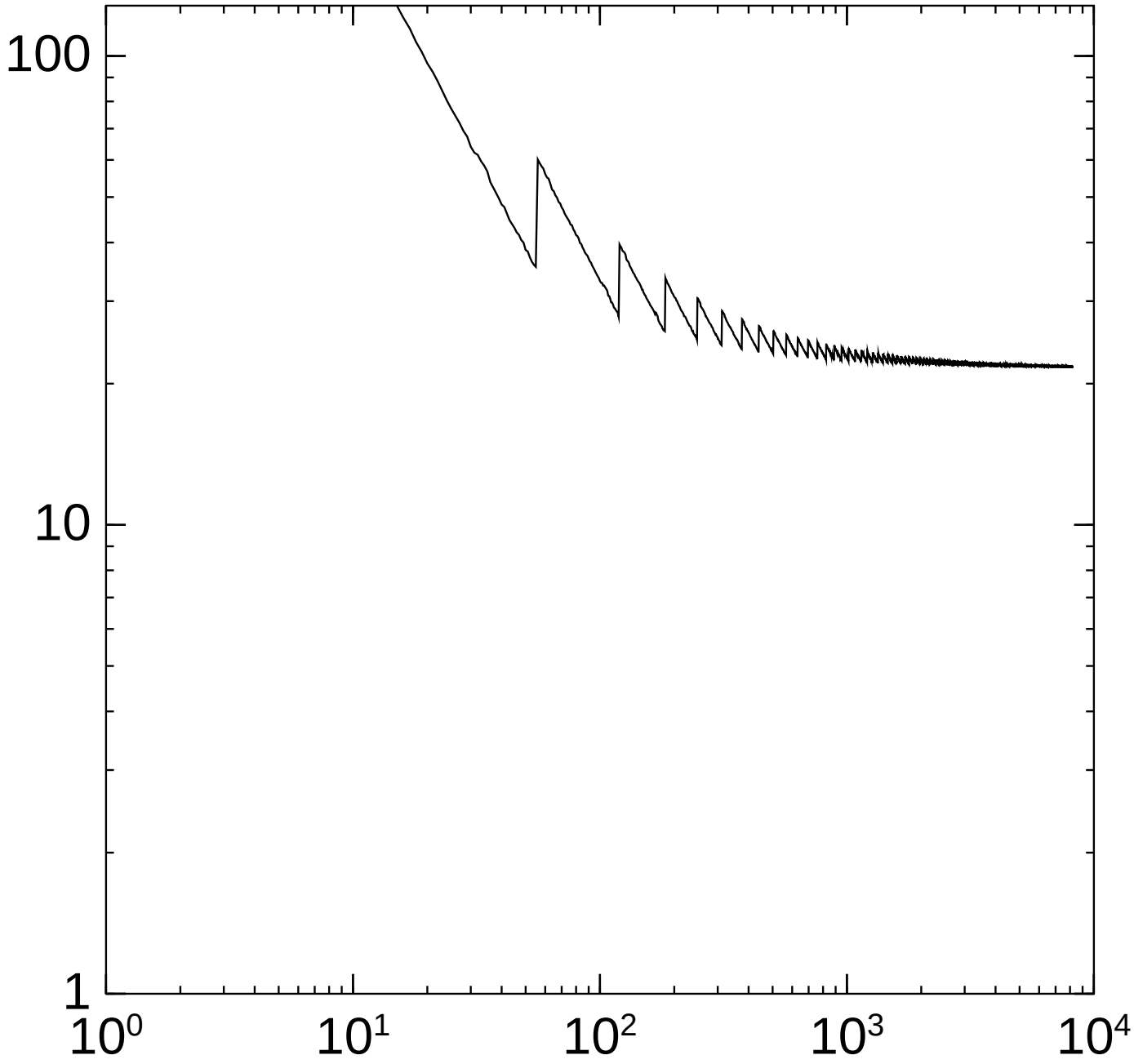
MD5 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



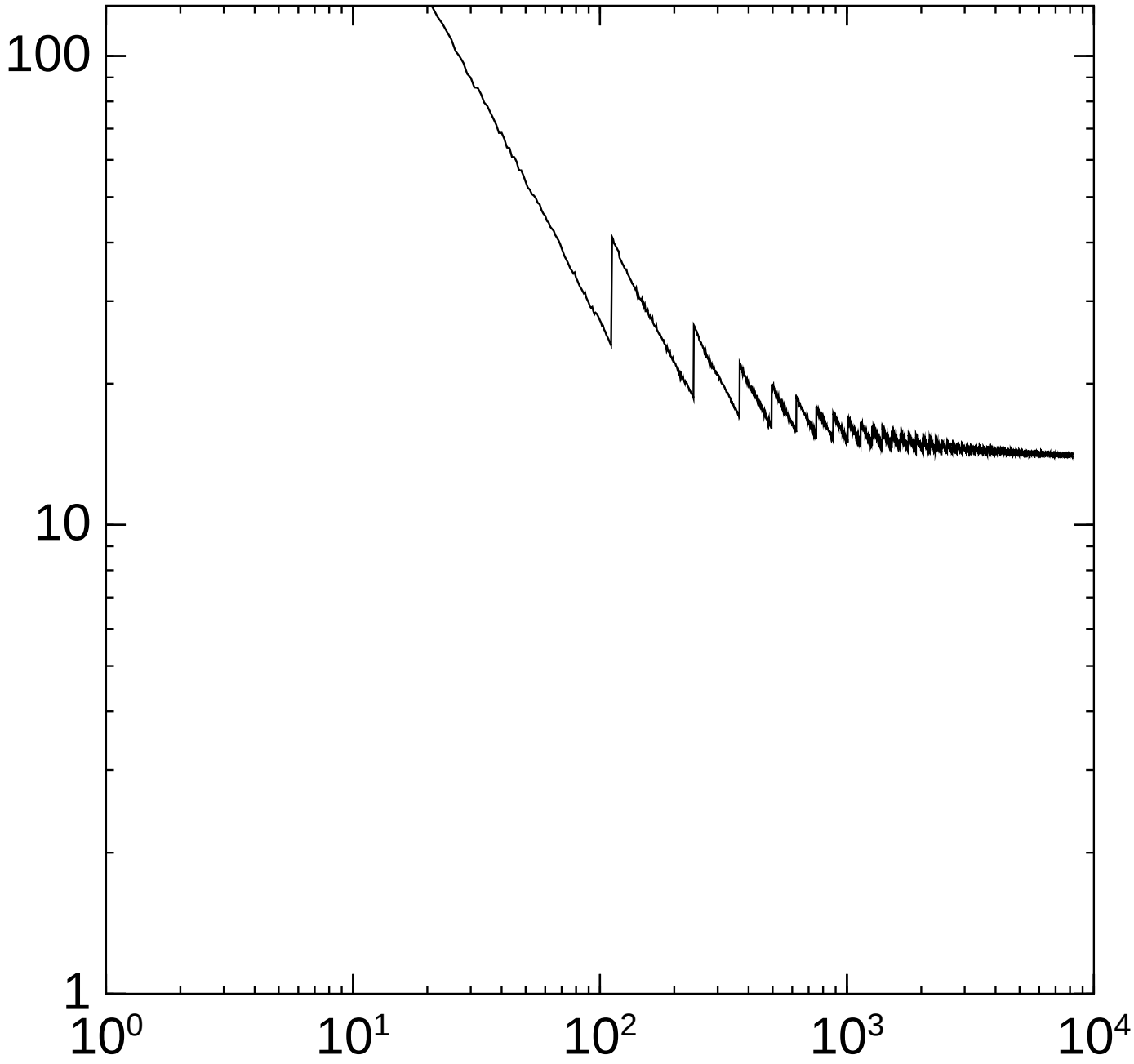
SHA-1 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



SHA-256 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



SHA-512 cycles/byte on `katana`, amd64 arch, Core 2 Duo:



Whirlpool cycles/byte on `katana`, amd64 arch, Core 2 Duo: