PPM Code Kernel Performance

Paul Woodward and David Porter
Laboratory for Computational Science & Engineering
University of Minnesota
February 3, 2005

Introduction

Thanks to a generous donation of an ES-7000 machine from Unisys to the LCSE, our team at the LCSE has had the opportunity to study PPM code kernel performance in detail on Intel Itanium-2 (IA-64) CPUs and to compare it with performance on Intel Pentium-4 and Pentium-M CPUs. The testing reported here shows that with appropriate effort one can obtain excellent performance on these Intel processors. The general coding techniques used to obtain this performance are described briefly here and in more detail in another document, a discussion of the PPM performance spreadsheet (www.lcse.umn.edu/PPM-performance-spreadsheet). In the study reported here, performance is first measured for PPM code kernels that update single strips of grid cells repeatedly, working from data that usually fits entirely into the processor's cache memory. These measurements give an idea of how the mix of instructions characteristic of modern fluid dynamics applications can execute essentially right "on the metal." Next, for a demonstration code, built in collaboration with Unisys using one of the kernels, a full two-time-step update of a single 2-D grid tile on a single CPU, with all its data reorganization and also with the generation of an image to visualize the developing flow, is executed. This allows us to measure the aggregate performance of the system on the mix of work that is actually necessary for a run involving user interaction (through the graphical display) as well as the application of boundary conditions and the many data copies, transpositions, etc., that are involved in a real world computation. Then for our two most used PPM code versions, a full single-time-step update of a single 3-D grid brick is executed, again with preparation of image data, in this case for use with our 3-D volume rendering utility, HVR (see www.lcse.umn.edu/hvr). Finally, 2-D grid tile and 3-D grid brick updates are used as building blocks for larger-scale parallel computation within our SHMOD framework (see www.lcse.umn.edu/shmod), which allows us to measure the efficiency with which the Unisys ES-7000 supports parallel computation.

The codes reported here achieve performance levels that are significantly higher than the average workload at, for example, NCSA. It is therefore reasonable to ask why they do this: is it something about the scientific problems they are addressing that permits this, or is it rather something about the way in which the numerical algorithms are designed and implemented? This is not an easy question to answer, and therefore in this paper we take the approach of explaining the key techniques in obtaining this performance at each level of the code structure – single CPU kernel, single CPU task, multiple CPU job on a single machine, and multiple machine job on a cluster. It is then left to the reader to decide what is special, the science addressed or the code design, or both. Today's computing systems are so complex that it is not possible to distill effective code design into just one or two simple principles. Overall performance is achieved through effort expended at each level of the computing hierarchy. However, this effort can be divided among people, as can the coding itself, so that it is not as onerous as it might seem. It is hoped that if one cannot apply all the techniques described below in one's own work, at least some of them may be used to improve performance at one or another of the levels of the code and computing system structure. We will begin our discussion at the level of a code kernel for execution on a single CPU out of its cache memory, then build outwards to a complete task performed by a CPU working out of its main memory, and finally to an entire job working from a memory shared either in hardware or in software by many CPUs.

Test Systems and Code Kernels

Three Intel CPUs were used for these performance measurements – the 1.7 GHz Pentium-M CPU, with 1 MB cache, in a Dell Inspiron 8600 laptop machine, the 3.2 GHz Pentium-4 CPU, Extreme edition (with 2 MB cache), in a Dell desktop PC, and the 1.3 GHz Itanium-2 CPU, with 3 MB cache, in a 16-processor Unisys ES-7000 machine. All 3 machines use the 800 MHz frontside bus, so that in principle their paths to main system memory run at the same bandwidth. The Unisys ES-7000 consists of two 16-processor SMPs, each with 32 GB of memory. Obviously, the large shared memory and 32 CPUs of the Unisys machine give it many advantages for large-scale parallel computation. Our parallel computing model uses a shared memory paradigm, so it operates most naturally on a system of this type. For cluster systems, we implement a global shared memory manually in software at the application, rather than at the system, level. This technique, which we call SHMOD, has been described elsewhere (cf. www.lcse.umn.-edu/shmod).

For the tests reported here, 5 versions of our PPM code have been used:

1. sPPM. *This is a simplified version of PPM that the LCSE team generated in 1996 for the DoE's ASCI program to use as a performance benchmark. It is publicly available in source code on our Web site (www.lcse.umn.edu/research/sppm). Many vendor representatives who have dealt with this code have been unaware that it was intended only as a benchmark code, not a real application. It is included here, because it is by now a well known item.* Two versions of this benchmark kernel are used:

   a. The first version is the fully vectorizable version originally demanded by the ASCI program. This is the version on the LCSE Web site.

   b. The second version is a scalar, pipelined version generated by hand and shared over the years with compiler teams from several different vendors. This version ran 58% faster than the vector version on the MIPS R-10000 CPU and 100% faster on the HP PA-RISC CPU 5 years ago. It has about 25 loops all combined into one by a process of aggressive loop fusion that has been implemented in some vendor compilers. The final, combined loop is then unrolled once for the benefit of superscalar microprocessors. It is an example of scalar code containing almost no cache misses, as a result of the loop fusion process (see www.lcse.umn.edu/HPDC8 appendix for statistics on the MIPS R-10000). It is truly scalar, since the Cray C-90 was brought by this code to the unprecedented low performance level of just 16 Mflop/s. This code kernel performs the identical arithmetic of the vectorized version, but in a totally different order. Today's microprocessors offering hardware support for vector computation out of their cache memories and supplied with compilers targeting this sort of vector execution do not prefer this version of sPPM. Nevertheless, it provides us with a useful estimate of the scalar-to-vector speedup on the Intel CPUs studied here, and it is thus included for reference.

2. PPMslowFlow. *This is a version of our present production PPM kernel that is suitable for slow flow applications (such as solar convection or perhaps mesoscale weather, for which shocks, if they occur at all, are so weak that the entropy they generate can be neglected). It is entirely vectorizable, like sPPM, but unlike sPPM this code is something we would actually care to run.*

3. MultiFluidPPM. *This is a new and highly complex version of our present production PPM kernel. It treats two-fluid flow problems with, potentially, strong shocks. It incorporates a very elaborate fluid interface treatment which is vectorized but which involves a great deal of vectorizable logic. This scheme updates 16 independent variables per grid cell in 3D.*

4. PPM. *This is our standard version of our present production PPM kernel. It treats only single-fluid flow, but it does so very carefully, with special work performed at contact discontinuities and at strong shocks. This scheme updates only 5 independent variables per grid cell in 3D.*

5. StateFairPPM. *This is a very complex version of our present production PPM kernel. It treats only single-fluid flow, but it very carefully advects 3 scalar fields, and it less carefully advects a fourth scalar field. In the state fair demo these 3 scalar fields are the R, G, and B color channels of an image. In more scientific applications, these might instead be chemical concentrations. They could also represent the original x-, y-, and z-locations of the fluid elements at some instant, so that a continuous distribution of tracer particles in the flow could be visualized to reveal its dynamics. The advection scheme used here has roughly 3 times the linear resolving power of PPM when applied on the same grid. This scheme updates 24 independent variables per grid cell in 2D.*

These code kernels are all, of course, gas dynamics schemes. Nevertheless they cover an extensive gamut, from the totally scalar sPPM code, to the relatively simple vector code of SlowFlowPPM, and finally to the extremely complex, only partially vectorizable code of MultiFluidPPM with its hundreds of vector temporaries (the Intel 7.0 compiler vectorizer claims that the main module of this last kernel "cannot be analyzed" unless scratch arrays are statically dimensioned; note that this "kernel," with comments, is several thousand lines of Fortran-77). The standard version of PPM was recently used to perform a simulation of homogeneous, Mach 1 turbulence on a grid of 2048^3 cells using a variable number, between 80 and 500, Intel Itanium-2 CPUs at NCSA (see www.lcse.umn.edu/shmod/IEEE-Cluster-Computing-paper-4-27-04-2cols.pdf and www.lcse.umn.edu/VDA2005). In all tests, even those for single, 1-D grid strip updates, all the kernels except the state fair kernel perform all the work that is required for a 1-D pass of a fully 3-D computation. The state fair kernel does the 1-D grid strip update from a full 2-D computation.

Code kernel N=4096	**Flops/Cell Update (1-D pass)**	**Adds/Cell Update (1-D pass)**	**Mults/Cell Update (1-D pass)**	**Recips/Cell Update (1-D pass)**	**Sqrts/Cell Update (1-D pass)**	**Cvmgms/Cell Update (1-D pass)**
sPPM scalar	701	255	264	23	8	68
sPPM vector	734	266	286	23	8	116
PPM SlowFlow	680	293	249	13	7	141
PPM StateFair	1345	630	644	19	1	386
PPM MultiFluid	1426	616	647	35.5	4	380
PPM Classic	921	430	409	17.2	2.2	249

This table gives the computational makeup of the 5 code kernels used in the performance tests. The two versions of the sPPM benchmark kernel actually perform identical arithmetic. However, in the scalar implementation, a few redundant computations have been eliminated upon fusing multiple calls to the same subroutine in the vectorized version. The multifluid PPM kernel is by far the most complex. It contains several scalar loops dealing only with cells in strong shocks, and it has the largest temporary workspace that needs to be held in the cache memory.

A Working Definition of the Flop

Although over the years there has been great temptation to redefine the flop, we have here chosen to follow established tradition in this study. The applications reported here all count their own flops. None relies on any hardware counters, since these have historically been based on varying definitions of the flop. It is the definition of the flop that was implemented in the Cray hardware performance monitor over two decades ago that here is taken to be the standard. As a result of manually programming the test codes so that they accumulate their own flop counts, there may be small errors in flop counting here, but the counts are at least uniform across the CPUs.

Operations are summed as the computation proceeds, accumulating totals in the categories of adds, multiplies, reciprocals, square roots, reciprocal square roots (usually implemented as a single operation on microprocessors), exponentials or logarithms, and “cvmgm” operations. The cvmgm operation (or its equivalent), so named by Cray (for “Cray vector merge minus,” although it is actually a vector mask rather than a vector merge operation), is the foundation of vector logic. It has also been termed a “conditional move,” and it is found as a basic instruction on most microprocessors. It examines the sign of a floating point number and based on the result selects one or the other of two floating point operands. Following the Cray tradition, we give no credit in the flop total to the cvmgm operations, despite the fact that they represent valid and useful labor. The add and the multiply each count as one flop, even if they are performed in a single multiply-add operation. A divide is counted as 4 flops, unless it is actually just a reciprocal, in which case it is counted as 3 flops. No modern CPUs made in America perform divides this fast, but we count them the way Cray did, since that is the standard. It is possible to implement divisions in terms of adds and multiplies and to count them as such. This yields much higher flop counts, and in this work this sort of counting is viewed as illegal. Sometimes comparisons count as one flop each, but only if we must first perform a subtraction in order to get a sign to examine. It is with the remaining operations, square roots, exponentials, and the like that we have problems defining flop counts. Here we have assigned 14 flops each to these operations, which past experimentation seemed to reveal as the choice made by Cray long ago.

We have counted the flops reported here, and programmed them in, following the rules set out in the previous paragraph. No redundant flops in common subexpressions are counted. The main ambiguity comes in vectorizable loops with logic. For example, there are loops in which fluxes at cell interfaces are computed using items from the cell on the left or on the right depending upon the sign of the time-averaged interface velocity. In a scalar implementation, there would be a single test on this sign and the arithmetic would be performed once, either with one set of values or with the other. These sections of code have been counted as if there were a *cvmgm* operation to select each operand. This procedure inflates the number of tests, but these tests do not count as flops. The arithmetic that is performed, one way or the other, is counted only once. It has been interesting to compare these flop counts with those generated by hardware monitors. In the case of the MIPS processors, the hardware count was low. In the case of the Intel IA-64 processors, the hardware count is high. The principal injustice of the flop counts reported here is that they attempt to adhere to an obsolete Cray standard that gives too little credit for the actual work performed in obtaining reciprocals and gives no credit at all for logical operations. For this reason, the counts in the various categories are broken out separately in the table above.

1-D Sweep Kernel Performance

The tables below give the performance of the 1-D kernels in Mflop/s. All these CPUs have rather large caches, so that performance degrades only at quite long vector lengths. In each performance test, a real 1-D problem is solved, so that any scalar loops involving execution in only a small, dynamically varying subset of the 1-D grid strip are exercised to a realistic degree. However, the 1-D grid strip updates are called multiple times at each time step level in order to greatly lessen the measured cost of boundary condition setting, copying of output to input arrays in cache, or any potential traffic to main memory. The performance measurements are therefore useful in assessing the effectiveness of these CPUs in performing computationally intensive tasks from their cache memories. Therefore these figures do not give an entirely accurate guide to performance that might be achieved in performing a real 1-D application (but such applications are rare today in any event). All the performance measurements show a roll off at large vector lengths caused by the failure of the workspace to fit into the cache memory.

32-bit PM	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	539	554	561	571	573	577	574	577
Vector sPPM	985	1237	1467	1591	1634	1600	1102	638
SlowFlow PPM	1205	1430	1511	1585	1588	1555	1465	780
StateFair PPM	751	877	1008	1100	1109	1031	772	648
MultiFluid PPM	947	1094	1193	1247	1250	919	648	563
Standard PPM	864	1031	1147	1230	1255	1144	746	593

Performance of the code kernels with 32-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

32-bit P4	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	683	698	705	734	739	740	742	741
Vector sPPM	1310	1666	2017	2263	2380	2242	2227	1794
SlowFlow PPM	1650	1907	2124	2078	2169	2069	1982	1896
StateFair PPM	957	1111	1280	1378	1399	1423	1412	1185
MultiFluid PPM	1203	1388	1523	1569	1570	1457	1332	1032
Standard PPM	1146	1346	1493	1577	1607	1530	1440	1159

Performance of the code kernels with 32-bit arithmetic on an Intel Pentium-4 (Extreme edition) CPU running at 3.2 GHz in a Dell desktop PC.

These results show performance for 32-bit arithmetic that is fairly consistent across these related processor families. The Itanium-2 processor targets 64-bit arithmetic, and therefore its performance for 32-bit arithmetic is only slightly faster, and in some cases can even be slower. It is interesting that the Pentium-M holds its own quite well against the Pentium-4, which here has nearly double the clock rate, but certainly not double the performance in 32-bit mode. We note that the codes run on the Pentium-M and Pentium-4 CPUs are compiled with the Intel 7.1 Fortran compiler, generating code exclusively for streaming SIMD extensions 2 (SSE2). This produces much faster code than use of the Intel 8.0 Fortran compiler, even if that compiler is directed to target each individual CPU exclusively. The 8.0 Fortran compiler also significantly reduces performance (by up to 30%) on the Itanium-2 CPU for some of the codes tested here (notably sPPM and State Fair PPM). In these cases the Intel 7.1 Fortran compiler is used. For the Itanium-2, the 8.0 compiler produces slightly faster code for the standard version of PPM in

32-bit mode, but significantly slower code in 64-bit mode. The 8.0 compiler does consistently produce superior performance for very long vector lengths, when the workspace of the calculation can no longer fit into the cache. Since many applications are not written to fit into the cache, such codes may have been a special target for the 8.0 compiler development. Perhaps compromises were made to enable compiling such code well that, unfortunately, reduced performance significantly for code of the sort studied here, where great effort is made to keep all computationally intensive kernels working within the cache memory. The behavior of the two compilers can be seen in the table entries for the code kernels for which results from both compilers are given.

32-bit Itanium-2	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	560	570	574	572	575	576	576	577
Vector sPPM (7.1; O3)	1065	1317	1516	1662	1717	1762	1772	1774
SlowFlow PPM	1216	1364	1452	1525	1538	1535	1532	1534
StateFair PPM (7.1; O3)	1016	1149	1265	1329	1363	1381	1379	1220
StateFair PPM (8.0; O3)	890	956	1014	1043	1082	1130	1146	1021
MultiFluid PPM (7.1; O3)	948	1013	1104	1161	1060	1062	1068	883
MultiFluid PPM	975	903	1017	1036	1020	1035	1164	973
Standard PPM (7.1; O3)	1001	1106	1176	1201	1215	1230	1227	1199
Standard PPM	1068	1160	1224	1240	1244	1251	1255	1220

Performance of the code kernels with 32-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in the Unisys ES-7000. Intel Fortran compiler versions and option flags are indicated in parentheses.

Either compiler for the Itanium-2 is used with the "/O3" compiler flag. The "/fast" compiler flag turns on interprocedural optimizations and perhaps some other things. These usually (but not always) tend to reduce performance for our codes when 32-bit arithmetic is used, while for 64-bit arithmetic they deliver some benefit. The multifluid PPM code kernel in particular has been manually optimized by inlining what were previously separate subroutines and by aggregating individual short loops, that the compiler demanded for automatic vectorization to be applied, into single, long loops preceded by vectorization directives. These loop aggregations always improve performance in these codes, although the compiler is apparently unaware of this fact. In these codes subroutine inlining never reduces performance, it sometimes is neutral in effect, and it can deliver an additional 50 Mflop/s or so in certain cases. The compiler appears not to make subroutine inlining transformation decisions properly on its own, at least for the rather large subroutines involved in these kernels. Therefore the inlining was done manually, which has an adverse impact on the code readability.

64-bit PM	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	412	432	441	443	447	448	448	448
Vector sPPM	481	517	538	544	538	414	281	242
SlowFlow PPM	652	677	698	715	698	660	365	298
StateFair PPM	545	592	625	644	604	428	350	279
MultiFluid PPM	552	569	619	623	476	336	289	184
Standard PPM	611	653	677	689	628	410	309	279

Performance of the code kernels with 64-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

64-bit P4	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	546	569	577	584	581	583	586	585
Vector sPPM	807	901	974	995	970	950	828	638
SlowFlow PPM	1115	1232	1297	1317	1290	1213	1151	726
StateFair PPM	900	985	1042	1063	1048	1030	769	685
MultiFluid PPM	922	971	1004	985	947	827	605	534
Standard PPM	1032	1087	1129	1132	1073	1040	728	620

Performance of the code kernels with 64-bit arithmetic on an Intel Pentium-4 (Extreme edition) CPU running at 3.2 GHz in a Dell desktop PC.

The kernels and codes used here had to be massaged so that the "/O3" compiler flag would result in generation of correct compiled code on the Itanium-2. The effort involved in making /O3 work was definitely worth the trouble. A helpful technique proved to be the designation of fixed lengths for work-space arrays, even if these were dynamically allocated on the stack. For the State Fair PPM kernel, one large workspace had to be statically allocated on the heap, via "save" instructions, in order for correct code to result with the "/O3" compiler flag. These fixed length working storage allocations required recompilation for each vector length, but compilation is very fast, and that was no particular trouble.

The Intel Fortran compilers try to make intelligent judgments about whether or not to vectorize a given loop. In a small number of cases, decisions by the compiler not to vectorize a loop give very slight performance improvements. However, it is much more common that these judgments by the compiler reduce performance for the codes studied here. Therefore directives were inserted to force vectorization for all vectorizable loops in all these codes. This procedure had the felicitous side effect of speeding up the compilation process enormously. It is a mystery why different vectorization assertions are required for the same compiler, depending upon the target Intel CPU. Nevertheless, both assertions were made, and the warnings generated by the wrong ones in either case were suppressed with the "/w" compiler flag.

Because these are cache-based CPUs, performance varies from one execution of the compiled code to another, even on a dedicated machine. The numbers quoted are the best observed. For the bulk of the table entries, the worst observed numbers were within 2% to at most 5% of the quoted values. Performance variation is greatest near the special vector lengths at which the entire workspace no longer fits within the cache memory, and this variation is much larger (up to 50%) for the 7.1 Fortran compiler in this special region than for version 8.0. Since long grid strips can easily be updated a section at a time, it is the performance when the entire workspace fits into the cache memory that is most important, and the 7.1 version of the compiler is the superior one in this regime for most of these codes. When we run our codes in production mode on many processors, we often find that one or two out of the 32 processors is running slower by up to 30%. We have therefore written our parallel job execution framework to sense this, kill that particular process, and then restart it. We find that if a process starts slow, it will always be slow, and if it starts fast, it will remain so. Because of the way in which our parallel job execution framework is designed (see www.lcse.umn.edu/shmod), it is not a problem to kill and restart these occasional slow processes.

64-bit Itanium-2	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048	N = 4096
Scalar sPPM	560	570	574	572	575	576	576	577
Vector sPPM (7.1; fast)	992	1178	1316	1325	1321	1294	1240	515
SlowFlow PPM	1123	1208	1241	1273	1255	1222	1202	1094
StateFair PPM (7.1; fast)	1076	1172	1212	1226	1231	1192	673	475
StateFair PPM (8.0; O3)	764	832	890	952	1000	1002	780	672
MultiFluid PPM (7.1; fast)	990	1040	1052	1060	1060	1039	598	461
MultiFluid PPM (8.0; fast)	779	853	936	988	1004	1009	803	634
Standard PPM (7.1; fast)	1126	1202	1230	1223	1214	1205	1106	405
Standard PPM	965	1037	1073	1091	1104	1108	1048	590

Performance of the code kernels with 64-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in the Unisys ES-7000. Intel Fortran compiler versions and option flags are indicated in parentheses.

We can see from the tables above that once we use 64-bit arithmetic, the Itanium-2 processor is working according to its design, and it delivers the best performance, even though it has the lowest clock rate of these 3 CPUs. Its peak performance is 4 flops per clock, or 5.2 Gflop/s. The 100% vectorized code of the sPPM kernel delivers 25% of this promise, which is exceptional, while the multifluid PPM kernel delivers 20%, which is still very good. This performance difference comes mainly from the small number of loops that are executed in scalar mode in multifluid PPM. Scalar loops are most efficient in performing the significant extra work that needs to be done at strong shocks, at centered rarefaction fans, and at any reflecting walls, because there are relatively few cells in which this extra work is required. Unless we are running the LinPack benchmark, we should not expect more than half the manufacturer's peak advertised performance. This is because the combined multiply-add instruction, responsible for half the peak performance figure, is nearly useless in normal computation. We note that the Pentium-M CPU does not support this instruction, but one would hardly know that from its performance numbers.

The performance numbers in the tables above do not take into account the cost of the process of preparing the data to be operated upon by the PPM kernel or in generating images from the results for the user to see. We would normally expect these ancillary processes to consume relatively little resources, however at the 6.2 GB/sec memory bus speeds in these machines, this is not quite the case, as the tables of the next section clearly show. In 2D, our PPM codes can run in interactive mode, presenting images of the evolving simulated fluid flows to the user as the computation proceeds. Using the 16 CPUs of the Unisys ES-7000 machine, quite detailed 2-D flow simulations can be performed in a fully interactive mode. Together with Unisys, our team at the LCSE demonstrated this at the Minnesota State Fair's Wonders of Technology exhibit in August, 2003. In the next section, we report performance measurements for the full 2-D grid tile update and image generation process from this interactive flow simulation application.

Techniques for Enhancing Single-CPU Kernel Performance

The Intel CPUs studied here are all cache-based microprocessors. Although CPUs of significantly different design are still available today, it is a general fact that code targeted to microprocessors like these will run nearly optimally on every CPU in the world today. The reason for this is that the Intel microprocessors involve in their hardware and compilers, in one form or another, nearly every successful strategy for improving code performance that has emerged over the last 30 years (with the exception, for the Itanium-2, of instruction reordering). Those features of the CPU design that have made achieving performance harder than on some other platforms have done only that: they have made our job harder, but they have not made it impossible. It may still be familiar to some that Cray vector computers of years long past delivered roughly half of their peak performance on well written codes (and of course everything delivers all of its performance to LinPack). It is not completely clear why microprocessors do not deliver 50% of their peak performance, but a likely reason is that they have very many fewer registers. If this is indeed the reason, it is clearly a cost issue, not only because registers are expensive but also because only scientists, who as a group are poor, seem to need them. Nevertheless, as we can see from the performance measurements for the PPM Slow Flow kernel, vector code on these Intel CPUs delivers 25%, rather than 50%, of the peak performance. Thus these CPUs are cost effective, even for scientists, so long as they cost less than half the price for equivalent vector CPUs. Occasionally, even scientists wish to go out of the vector paradigm to do some unusual computation, and then these microprocessor CPUs deliver half to one third of their vector performance, which is very good, as our table entries for the scalar sPPM kernel clearly show.

The code kernels tested here are all written in a single style, which is largely responsible for their high levels of performance. Each kernel targets the processing of a linear array of entities, in these cases a single, 1-D strip of grid cells. The data needed for this processing is handed to a subroutine as a collection of 1-D arrays. Even if the data actually comes from multi-D arrays, from the point of view of the Fortran subroutine doing the processing all the subroutine arguments are interpreted as 1-D arrays. There are no routines that are written to process only a single grid cell, since this would not allow performance enhancements through vectorization. Upon the call to the routine responsible for the grid strip update, a workspace consisting of tens to many tens of 1-D scratch arrays of equal length is dynamically allocated on the subroutine's stack. For a single-CPU machine, like a desktop PC or a laptop, the distinction between the stack and the heap (heap allocation is obtained by using Fortran "save" instructions) is minimal. On a multiprocessor, shared memory machine the distinction can be major. Since the stack is reallocated on each call, the compiler can take care that this memory is always allocated in the best possible place. On machines with many processors, this can be extremely useful.

Our code kernels all consist of unit-stride loops that process vector inputs and produce vector outputs. The fewer the number of vector inputs and outputs in relation to the amount of processing in the loop, the greater are the opportunities for the CPU to use its registers rather than its cache memory, and therefore the higher is the delivered performance. Vectorizable loops have highly predictable patterns of functional unit usage, and hence can be scheduled optimally. All the Intel CPUs studied here have special hardware instructions supporting vector computation. This allows groups of words to be loaded in blocks and multiple functional units to operate simultaneously without the possibility of conflicts. Automatic vectorization by the Intel compilers works only for relatively short loops, which cannot be fully efficient because they are forced to output temporary results of the computation in vector form for consumption by the following loop. Longer vector loops in general have higher ratios of computation to the sum of their inputs and outputs, which makes it possible for a good compiler to generate more efficient vector code from them (a possibility that is not always realized, but which can always be hoped for). The Intel compilers will agree to vectorize long, and potentially very efficient vector loops if the user inserts a directive just before the loop, which is highly recommended. Nonvectorizable loops can usually be written in the following format: the first statement of the loop is a test on a flag vector, which decides whether or not to execute the loop body. The flag vector is computed in a preceding vectorizable loop at high speed. Even with a vectorization assertion in front of such a loop, the Intel compilers will not generate vector code, so that this loop structure cannot be mistaken even if one simply puts vector assertions in front of all loops in the program. Since the vector-to-scalar execution speed ratio is about 3 to 1, such scalar loops should be used when one is fairly sure that no more than about a third of the flag entries are "true." It is almost always easy for the programmer to know which kind of loop is preferred, either vector or scalar, and it is then unnecessary for the compiler to make this decision.

One might ask how one can come up with the groups of 1-D, unit-stride vector inputs to computational kernels when one has multi-dimensional problems to solve. First, we note that it is possible to copy these vector inputs out of higher dimensional arrays explicitly. This is not needless copying, which might be expensive. The reason is that the vector computation we want to do demands short arrays that are resident in the cache memory. This data must therefore be copied into the cache in any event, whether we explicitly express that in our program or not. A typical modality is for a subroutine to operate on 1-D vectors that are refilled again and again with different data by the calling routine. If the entire workspace of the subroutine fits into the cache, then the data in these arrays need never, or almost never, be sent to the main memory. This is the advantage of a "write-back" cache; if we keep refilling the arrays, we can truly think of them as cache resident, with effectively no copy maintained in the main memory. The main memory is very slow, and this technique avoids using it at all for the bulk of the temporary storage required during the computation. The impact on code performance when all the temporary workspace cannot fit in the cache is easily seen in the performance tables of the preceding section.

From the above discussion it should be clear that it is highly worthwhile to keep the size of our temporary workspace small. One way to do this is to use short vectors. However, this technique usually comes at a cost in redundant computation. Another method is to reuse our temporary vectors that we have allocated on the stack. The first way to do this is to rely on the compiler to assist. In the process of vectorizing our code, the compiler will probably be forced to allocate temporary vectors to hold results for which there is not enough room in the registers. The advantage of having the compiler do this as opposed to doing it ourselves is that the compiler can reuse the same few vectors without becoming confused by all sorts of different quantities that are all given the same names. From this perspective, it was a step backwards for the Intel compiler team to force programmers to allocate such vectors in order to make use of vectorizable logic. A programmer can reuse vector temporaries that are explicitly allocated by employing the Fortran "equivalence" statement. This is a very dangerous programming technique. The same effect can, however, be achieved by breaking a subroutine into several ones to which temporary workspace vectors are passed as arguments. Then inside each subroutine a meaningful and readable name can be assigned to each temporary vector, and these names can be different in the different subroutines. Unfortunately, the Intel compilers will not inline these subroutines, thus generating the equivalences that were initially desired. Instead, performance will suffer, presumably because assumptions about the alignment of the vector arguments (with respect to multi-word boundaries) cannot legally be made, even if the interprocedural optimizations are turned on. Compiler assistance in this area would be very helpful. In its absence, one is forced to use too large a workspace if one wants a readable and maintainable code. Happily, caches of 1 MB or greater are available even on laptop processors today; nevertheless, managing the workspace size can be expected to be an important performance issue for some time to come.

Aside from avoiding divisions and special function calls wherever possible and avoiding use of short subroutines (manual inlining), the remaining technique that can deliver a major performance benefit for code kernels is the use of 32-bit arithmetic. This is discussed briefly below.

Why 64-bit Arithmetic?

Most computational scientists are convinced that they need 64-bit arithmetic. Nevertheless, the PPM codes studied here do not. It is not completely clear why people believe 64 bits are necessary, but the following theory might help to understand this. First, high arithmetical precision can allow convenient algorithmic expression. The elaborate advection scheme used in the multifluid and state fair PPM codes can help to illustrate this point. This advection scheme is derived from the PPB scheme described in Woodward 1986, which is a 2-D extension of Bram van Leer's Scheme VI, first described by van Leer in the 1970s. It updates not only average values of the advected function in grid cells but also moments of that function in the cells. The x- and y-coordinates used to generate the moments are scaled separately for each grid cell, so that the origin is at the center of the cell and the cell width is unity. Were the global x- and y-coordinates to be used, we would need another 3 or 4 digits of arithmetical precision on fine grids, and 32-bit arithmetic would probably no longer suffice. It is more trouble to express the algorithm in these cell-centric coordinates, but the algorithm is then better suited to computation. With 64-bit arithmetic, we could take the easier and more natural approach, but the cell-centric coordinates give us the option of faster computation on microprocessors with the use of 32-bit arithmetic at essentially no cost in solution accuracy. This is just one of many examples; unnecessarily demanding a global view raises a cost in required arithmetic precision. This can especially be seen if one demands that a global set of grid point values be coupled together by a matrix inversion process. An alternative formulation might be to iteratively solve a relaxation process based on just local data, with no requirement beyond 32-bit precision. Not everything that we sometimes assert to be mathematically coupled together is actually physically coupled at a practical level of significance. The trade-offs here do not only involve computational speed but also memory size requirements, both of cache and main memory. Use of 32-bit arithmetic effectively doubles both the memory size and the memory bandwidth of a computer. Both of these are very expensive commodities.

2-D Grid Tile Update Performance

We here report results of the 2-D state fair PPM code running on a single CPU. Results of several CPUs running this code in parallel are reported in a later section. According to the PPM parallel performance model, so long as all interprocessor communication is thoroughly overlapped with this computation, essentially all overheads associated with parallel computation of this job are accounted for within the single grid tile update. In addition to the cost of the raw computation using the PPM kernel code, the measurements below include all the costs of transposing, uninterleaving, re-interleaving, and possibly re-transposing the 24 vectors of fundamental fluid state quantities (such as the density, pressure, and velocity) that this PPM kernel processes for each grid strip. These data manipulations do not run at the raw main memory bandwidth, because the CPU must always rearrange the data as it streams into the cache so that contiguous, unit-stride vectors are formed in the cache to enable vector processing. Also, the results of the kernel computation must also be rearranged, so that all the values referring to the same cell are stored in contiguous locations in the main memory. This allows the data to be transposed efficiently upon the next 1-D pass of the algorithm.

32-bit 1.7 GHz PM	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Adds/Cell	3024	2051	1650	1469	1384	1343
Mults/Cell	3010	2049	1649	1470	1386	1345
Cvmgms/ Cell	1811	1235	996	889	839	814
Recips/ Cell	95.5	64.7	52.0	46.3	43.6	42.3
Sqrts/Cell	5.4	3.5	2.7	2.3	2.2	2.1
Flops/Cell	6398	4343	3493	3112	2931	2844
Sec/Δt	0.00508	0.0186	0.0555	0.203	0.833	3.58
Sec/Δt @N=4096	83.2	76.3	56.8	52.0	53.3	57.2
Mflop/s	647	716	778	752	692	626
Fraction 1D Perf.	86%	82%	77%	68%	62%	61%

Performance of the 2-D state fair PPM code with 32-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

64-bit 1.7 GHz PM	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Sec/Δt	0.0112	0.0297	0.0982	0.384	1.61	7.12
Sec/Δt @N=4096	184	122	101	98.2	103	114
Mflop/s	440	450	437	398	358	314
Fraction 1D Perf.	81%	76%	70%	62%	59%	73%

Performance of the 2-D state fair PPM code with 64-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

32-bit 3.2 GHz P4	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Flops/Cell	6398	4347	3492	3112	2931	2843
Sec/Δt	0.00578	0.0140	0.0412	0.141	0.538	2.26
Sec/Δt @N=4096	94.7	57.5	42.2	36.2	34.5	36.1
Mflop/s	850	951	1042	1083	1071	990
Fraction 1D Perf.	89%	86%	81%	79%	77%	70%

Performance of the 2-D state fair PPM code with 32-bit arithmetic on an Intel Pentium-4 CPU (Extreme Edition) running at 3.2 GHz in a Dell desktop PC.

64-bit 3.2 GHz P4	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Flops/Cell	6398	4347	3492	3112	2931	2843
Sec/Δt	0.00659	0.0170	0.0538	0.202	0.849	3.46
Sec/Δt @N=4096	108	69.6	55.0	51.6	54.3	55.4
Mflop/s	749	785	798	758	679	647
Fraction 1D Perf.	83%	80%	77%	71%	65%	63%

Performance of the 2-D state fair PPM code with 64-bit arithmetic on an Intel Pentium-4 CPU (Extreme Edition) running at 3.2 GHz in a Dell desktop PC.

The large cost of the data manipulations needed to prepare grid strips for processing in the cache memory is apparent from the tables above. Not only does this work take up 15% of the time, even for the smallest grid tile and 32-bit arithmetic, but as we try to make the computation more efficient by enlarging the grid tile, the temporary arrays that must be cache resident for the data manipulations to be efficient begin to crowd out the workspace of temporary vectors needed to update a grid strip. The result is that for the most efficient grid tile size, with N=256, only 2/3 of the time is spent computing. The remaining third of the time is spent rearranging data in memory in complex but highly ordered ways. Using 64-bit arithmetic, the situation is even worse. For the Pentium-4 and the Itanium-2 CPUs, the grid strip update is nearly as fast with 64-bit arithmetic, but the data manipulations are twice as slow. There is no reason why these data rearrangements could not proceed in parallel with the grid strip update, but this would require, with present technology, two threads sharing the single CPU. Even with Intel's concept of "hyperthreading" on a single CPU, it is not clear that the performance improvement would be worth the very significant programming effort. However, as CPUs continue to speed up and memory speed continues to lag behind, it may prove useful to contemplate such programming mechanisms. We note that during the data manipulations, as presently implemented, the floating point units are entirely idle. Language or compiler assistance with such asynchronous prefetching of large blocks of data into the cache would certainly be helpful. It should also be clear that this problem only gets worse when we go to 3D problems.

32-bit 1.3 GHz It2	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Flops/Cell	6398	4347	3492	3112	2931	2843
Sec/Δt	0.00567	0.0143	0.0431	0.145	0.600	3.47
Sec/Δt @N=4096	92.8	58.6	44.1	37.2	38.4	55.5
Mflop/s	867	934	997	1053	960	645
Fraction 1D Perf.	85%	81%	79%	79%	70%	47%

Performance of the 2-D state fair PPM code with 32-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in the Unisys ES-7000.

64-bit 1.3 GHz It2	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024
Flops/Cell	6398	4343	3492	3111	2931	2844
Sec/Δt	0.00647	0.0166	0.0506	0.198	0.934	4.38
Sec/Δt @N=4096	106	67.9	51.8	50.7	59.8	70.1
Mflop/s	759	805	848	772	617	510
Fraction 1D Perf.	71%	69%	70%	63%	50%	43%

Performance of the 2-D state fair PPM code with 64-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in the Unisys ES-7000.

The 2-D codes were compiled with /O3, while the 1-D ones were compiled with /fast, which produced a faster result. For the 2-D codes, however, /O3 made much faster code (20%).

Techniques for Enhancing Single-CPU Task Performance

The performance measurements in the tables above correspond to single CPU tasks. In each such task an entire grid tile with dimensions $N\times(3N/4)$, augmented by a picture frame of ghost cells 16 cells wide, is updated for two full time steps. The ghost cells are used to provide the proper boundary conditions for the grid tile update. For this algorithm, 8 ghost cells are required for each time step update. By providing 16 ghost cells, we are able to perform four 1-D sweeps on the data without any need to communicate with any other processor that is updating any other grid tile. There is a trade-off here between the cost of the redundant computation that is performed in the ghost cell regions of adjacent grid tiles and the benefit of uninterrupted, independent computation. One aspect of this trade-off is measured in the row of each table entitled "Sec/Δt @N=4096." These table entries tell us how much time would be required to update a 4096×3072 grid tile for 2 time steps, updating one subdomain of the given size at a time. In all but one case, using subdomains of 256×192 would give the best performance. This subdomain size does not always yield the highest Mflop/s, but the ratio of redundant to unredundant flops is quite good for this size. In the row of the tables labeled "flops/cell," the cost of all flops, both redundant and not, is given. This cost is lowered by only 6% in going to a 512×384 tile size, while the CPU performance usually decreases by more than a compensating amount.

The optimal tile size of 256×192 cells requires at least 0.14 seconds for a 2-time-step update. This is a reasonable amount of time for the CPU to be occupied with a single task, because it is much larger than any latency in the computing system, be it a network communication or a disk access. The slowest data bandwidth that is likely to be relevant to this computation is a data transfer over Gigabit Ethernet, which would proceed at about 35 MB/sec. During the 0.14 sec grid tile update, such a transfer could deliver nearly 5 MB of data. The ghost cell data for the grid tile is only 24 (variables/cell) × 4 (bytes/variable) × ((288×224) – (256×192)) cells = 1.9 MB. We could therefore both send and receive this amount of ghost cell data over such a network during the time it takes to update another grid tile. If we get the ghost cell data we need for our next grid tile while we are updating the present one, the calculation will not be slowed by waiting on data even if it is distributed over a (slow) Gigabit Ethernet network. On a Myrinet network, where transmissions at 150 MB/sec are possible, we might choose not to plan ahead in this way and to pause our computation while fetching this data. This choice would only slow us down by about 20%.

On a shared memory machine like the Unisys ES-7000, we can fetch the ghost cell data from shared memory at over 2 GB/sec. This is so fast that we can fetch the next entire grid tile, a record of just under 6 MB (only 3 times larger than just the ghost cell data). This frees our CPU from having to update one of some pre-assigned set of grid tiles, and it makes our computation much more flexible. If very different times are required to update different grid tiles, as is the case in many applications, when we finish a grid tile update we can fetch the next tile on some globally ordered list. In this fashion the 16 CPUs in the ES-7000 can cooperatively update the entire grid without any of them needing to wait for other CPUs to complete their jobs. When no more grid tiles need to be updated for this time step, if the global ordering of updates was done intelligently, it is almost certain that a CPU needing work can begin updating the first listed tile for the next time step. For the PPM codes, we need to compute a value of the time step interval to use, and this is properly the result of a global reduction operation. However, one time step usually resembles the next pretty closely, so we can obtain a speculative value for the time step using the best information available and go on. If the speculation proves incorrect, which is very rare, we can repeat the necessary grid tile updates with no problem.

Our SHMOD framework implements this parallel computation model for 2-D runs. SHMOD stands for Shared Memory On Disk. For very large 3-D runs we place all the grid brick records on disk. Since 150 MB/sec can be delivered to a set of striped SATA disks on a PC, there is more than enough time to prefetch the next grid brick while updating the present one. This allows small machines to perform huge problems. We have used this technique to do a PPM simulation on a 1024^3 grid on our ES-7000, as is discussed in a later section. For 2-D problems, where fitting the entire data context into the ES-7000 memory is not a problem, we can nevertheless use the same SHMOD paradigm by storing the grid tile records in a memory-mapped file or files, which can be accessed by all the CPUs. Then each CPU need only read the separate "disk" records for the interior, ghost cell edges, and ghost cell corners of its next grid tile from the memory mapped file in 9 sequential data transfers. The SHMOD framework composes these separate data records into a single one. A single array, DD, dimensioned as follows with the first, fast-running index going over the 24 physical state variables in each cell, DD(24,-15:nx+16,-15:ny+16), holds the data to be updated. This composition process for the array DD goes at over 2 GB/sec, so that its cost is negligible. The performance data in the tables above report the speed at which the Intel processors studied here update the data in DD.

We have already discussed the manner in which each grid strip update is implemented, and why that implementation makes sense on this hardware. The 2-D grid tile update, starting from the array DD, consists of a sequence of calls to the 1-D grid strip update routine. To prepare the data for these calls, we need to construct the set of 1-D grid strip vectors that the 1-D routine expects, and we must take the 1-D vectors it produces and place them back into a new version, DDNEW, of our DD array. In the x-pass, this involves reading entire rows of DD data and sprinkling the results into individual row vectors in the cache memory, with one vector for each of the 24 physical state variables. This process goes quite rapidly, at just under 2 GB/sec on the tested hardware. After updating the grid strip, the 24 new vector results must be interleaved for storage into DDNEW in the main memory. This is also quite rapid. In the y-pass, however, we need to effectively transpose the data in the DD array. This is not straightforward, because of the organization of the main memory in cache lines. To get good memory performance, we must read in several cells worth of data in a row. Here the number of cells we need to read is reduced by a factor of 24 because we have interleaved all the state variables in the single array DD. We have found that it is a good idea to read in groups of 4 to 8 cells in the same row of DD at a time. This not only gives good memory performance, but, as we will soon see, prevents the need to read the data over and over again. Speeds of up to 1.5 GB/sec can be obtained on the tested hardware for this process, but not if all the data that is read does not fit into the cache.

Anticipating our 3-D applications, we will call the band of grid strips that we read into the cache as a grid pencil. The PPM algorithm requires data in two grid strips above and below the present one in order to perform its update. This means that if we are to update a pencil of 4 grid strips, we need to read 8 grid strips into the cache. In principle, we could reuse some of this data for processing the next grid strip. However, we have written the code so that the next grid strip might actually be updated by another CPU sharing the same main memory. This means that if we update pencils of 4 grid strips, we end up reading each data value in the DD array twice, effectively halving our CPU's main memory bandwidth for this operation. By updating pencils of 8 grid strips, we reduce this overhead by a factor of 2. This consideration is most important in 3-D, where updating a 4×4 grid pencil involves reading into the cache an 8×8 set of grid strips.

The substantial cost of the data rearrangements just discussed is clear from the performance data in the tables above. The large number, 24, of state variables and the 8-strip grid pencils used take up a significant portion of the cache memory available, especially when we use 64-bit arithmetic. The Itanium-2 processor in the Unisys ES-7000 is the top performer here, at 772 Mflop/s. The cost of the memory manipulations, however, means that this is only 63% of the computational performance possible for a single grid strip update. Thus, 47% of the 0.198 seconds required for the tile update are spent rearranging memory. The total amount of data read from and written to main memory in this process, over 4 1-D passes of the algorithm, is $24\times8\times(288\times224 + 276\times220\times2 + 272\times208\times2 + 268\times196\times2 + 256\times192)$ bytes = 84.1 MB. However, we did not account for the fact that by reading grid pencils 8 strips wide, we must actually read half of the data twice, although we only write each datum once. This increases our estimate by 25% to 105 MB. In addition, at the outset of the grid tile update, our test code constructs the grid tile from an interior, 4 ghost cell edges, and 4 ghost cell corners, applying boundary conditions in this process. At the end of the 2 time step update, these 9 individual records are copied out from the new DD array. These copies each involve both reading and writing the data, so that they add another $4\times24\times8\times288\times224 = 47.25$ MB to our total. Thus our total becomes 152 MB. Assuming that the generation of the image, which is part of the cost of the grid tile update, is negligible (which we have verified), then the average delivered memory bandwidth for these data manipulations is $152/0.0931 = 1.6$ GB/sec. At well over half the speed we could expect for a simple, sequential data copy, this average data manipulation speed is quite respectable. The memory bus bandwidth is 6 GB/sec, but our data manipulations are far more complex than simple sequential copies into or out of the cache. Had we not interleaved our state data in a single DD array, our results would have been much worse.

The relationship between numerical algorithm accuracy and memory bandwidth requirements:

The above discussion shows that the PPM state fair application has a memory bandwidth requirement that, with 64-bit arithmetic, forces nearly half the time to be spent reading and writing data to and from main memory. This is because it updates 24 state variables in each grid cell with 1345 flops in each of its 1-D passes. In contrast, the standard PPM algorithm updates only 5 state variables with 921 flops. These facts are best stated in terms of the bytes/flop ratio. For the state fair PPM 1-D pass, 384 bytes are transferred for every 1345 flops, with 64-bit arithmetic, giving a ratio of 0.286 bytes/flop. For standard PPM we get 0.087 bytes/flop, which is much better. When these algorithms work out of the CPU's cache memory, the impact of these very different bytes/flop ratios is negligible, since far more bandwidth than either algorithm requires is provided by the hardware. But in working from the main memory, the factor of 3 difference is very important.

Here we see a general principle at play. The more work we can perform on local, cached data with our numerical algorithm, the better the machine performance will be. Decreasing the bytes/flop ratio to enhance machine performance is not difficult. There are two general techniques. Both involve increasing the accuracy of the computation, which is good in and of itself. The first technique, which is the easiest, is to build a more accurate grid cell update by subdividing the cell into quadrants (in 2-D) and updating these for 2 half-time-steps in a single operation. This operation requires 4 times the data to be transferred, but it does 8 times as much arithmetic. Hence its bytes/flop ratio is half that of the original process. We note that in the state fair grid tile update we are effectively doing this already, since we are performing 4 1-D sweeps on the same data. However, we have not organized the computation in such a way that all 4 sweeps are performed at once while a particular subset of the data remains in the cache memory. This could in fact be done by using a general programming technique which we call aggressive loop fusion. We would need to formulate our 1-D sweep algorithm so that it processed rows rather than columns of data at a time in our y-passes. In such a formulation, we would need to have a separate row vector for, say, the density in our cells of interest and for the densities in the cells below them. We would need to sweep through our grid row-by-row, and before we could produce a row of entirely new state variables, updated through all 4 1-D sweeps, we would have to partially process a great many rows (26 in fact) above the one for which the updated results would finally become available. Performing this code transformation by hand is tedious and prone to error, but compilers can do this for us. We have worked with Rob Fowler and the compiler team at Rice University to experiment with this procedure using Bob Wilhelmson and Lew Wicker's NCOMMAS meteorology code with some success.

A second, less straightforward but perhaps more rewarding technique for increasing the bytes/flop ratio is to use a fundamentally more accurate difference scheme. There is a general relationship between the accuracy of a difference scheme and its computational intensity. Here we mean accuracy in a practical rather than in a formal sense. In the state fair PPM code, 3 fields are advected with a moment conserving difference scheme that is, in 1-D, formally 5th-order accurate. In 2-D the formal accuracy is much less, but as a practical matter this advection scheme delivers about the same accuracy as the PPM advection scheme operating on a grid refined by a factor of 3 in each spatial dimension and time. This scheme performs about twice as many flops as PPM advection operating, in 2-D, on 6 times as much data. But to achieve this same accuracy with PPM advection, we would need to operate on 9 cells worth of data. Hence, for equal quality results, the bytes/flop ratio of the state fair code's advection scheme is smaller than that of PPM by a factor of about 3, even without accounting for the 3 time steps we would need to take with the PPM advection scheme on its imagined subdivided grid. The computational intensity of the PPM gas dynamics scheme itself also is related to its accuracy. Were we to subdivide its grid and perform a much simpler updating scheme for the subcell patches, we could equal its accuracy, but the bytes/flop ratio would be very much larger. In a sense, the point we are making here is that on modern microprocessor CPUs we can think of computation as free when compared to data movement. If there is any reasonable additional computation we can perform to increase the accuracy of our result using the same cached data, we should not hesitate to do it.

3-D Grid Brick Update Performance

We here report results of the 3-D standard and multifluid PPM codes running on a single CPU. Results of several CPUs running these codes in parallel are reported in a later section. As with the 2-D tests reported above, the PPM parallel performance model implies that, so long as all interprocessor communication is thoroughly overlapped with this computation, essentially all overheads associated with parallel computation of a 3-D job are accounted for within the single grid brick update. In addition to the cost of the raw computation using the PPM kernel code, the measurements below include all the costs of transposing, uninterleaving, re-interleaving, and possibly re-transposing the 5 or 16 vectors of fundamental fluid state quantities (such as the density, pressure, and velocity) that these two PPM kernels process for each grid strip. Data for producing volume rendered 3-D images is generated upon each time step as well. As in the 2-D case, the cost of the image data generation is negligible. However, the 3-D data manipulations consume a greater fraction of the time than in 2-D. Grid pencils of 8×8×N cells are constructed in the cache memory for each fluid state variable from data stored in a single 4-D array for which the first dimension (the fast-running index) is the number of the fluid state variable. Thus, as in the 2-D case, we store the variables for each cell together in main memory in order to allow efficient transposition of the data for the y- and z-passes of the algorithm. The cost of the necessary data manipulations is significantly greater in 3D than in 2D, which is reflected in the last rows of the tables below. In those rows are entered the fractions of the 1-D grid strip update performance that are obtained by the full 3-D applications.

32-bit 1.7 GHz PM	N = 32	N = 64	N = 128	64-bit 1.7 GHz PM	N = 32	N = 64	N = 128
Flops/Cell	6116	4088	3455	Flops/Cell	6116	4080	3328
Sec/Δt	0.263	1.23	7.60	Sec/Δt	0.386	2.01	13.1
Sec/Δt @N=512^3	1077	628	486	Sec/Δt @N=512^3	1582	1029	836
Mflop/s	762	873	954	Mflop/s	519	532	534
Fraction 1D Perf.	88%	85%	83%	Fraction 1D Perf.	85%	81%	79%

Performance of the 3-D standard PPM code with 32- and 64-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

32-bit 3.2 GHz P4	N = 32	N = 64	N = 128	64-bit 3.2 GHz P4	N = 32	N = 64	N = 128
Flops/Cell	6222	4253	3455	Flops/Cell	6240	4252	3444
Sec/Δt	0.200	0.952	5.69	Sec/Δt	0.228	1.19	7.70
Sec/Δt @N=512^3	820	488	364	Sec/Δt @N=512^3	933	608	493
Mflop/s	1019	1171	1273	Mflop/s	898	939	938
Fraction 1D Perf.	89%	87%	85%	Fraction 1D Perf.	87%	86%	83%

Performance of the 3-D standard PPM code with 32- and 64-bit arithmetic on an Intel Pentium-4 CPU (Extreme Edition) running at 3.2 GHz in a Dell desktop PC..

32-bit 1.3 GHz It-2	N = 32	N = 64	N = 128	64-bit 1.3 GHz It-2	N = 32	N = 64	N = 128
Flops/Cell	6222	4253	3455	Flops/Cell	6225	4252	3444
Sec/Δt	0.206	1.09	6.78	Sec/Δt	0.253	1.28	8.34
Sec/Δt @N=512^3	843	557	434	Sec/Δt @N=512^3	1037	655	534
Mflop/s	990	1025	1068	Mflop/s	807	871	866
Fraction 1D Perf.	93%	88%	87%	Fraction 1D Perf.	72%	72%	70%

Performance of the 3-D standard PPM code with 32- and 64-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in a Unisys ES-7000.

The results reported in the above tables for the standard PPM gas dynamics code in 3D are quite encouraging. As we make our grid brick bigger, the fraction of the 1-D grid strip update performance that we obtain declines due to the increasing cost of the manipulation of the data in memory, but none of the values here show this cost to be excessive. Grid bricks larger than 128^3 cells are not tested here. Such large grid bricks are useful in parallel computation of only extremely large problems, which can take months to complete. Using 32-bit arithmetic, we obtain about a gigaflop from each of these 3 processor types, and the cost of the data manipulations in the memory is no more than 17%. However, the multifluid PPM code, which updates 16 rather than 5 variables per cell and which performs only about 50% more flops for each cell update, has a bytes/flop ratio that is about two times worse than standard PPM. Not only is the cost of the data manipulations for the multifluid code greater due to this bytes/flop ratio, but the grid pencils of fluid state variables require about triple the storage space in the processor's cache memory for this code. The result is that the overall cost of the data manipulations is a little more than double that for the standard PPM code, as is clear from the tables below. This brings the overall performance below the 1 Gflop/s mark on all 3 types of CPU, but the performance is nevertheless quite respectable.

Using 64-bit arithmetic on the Itanium-2, updating a grid brick of 128^3 cells with the multifluid PPM algorithm, we obtain 615 Mflop/s, or only 12% of the processor's peak performance. In this case only 58% of the time is spent computing, and the remainder of the time is spent rearranging the data in memory. It is not clear how this situation can be alleviated, since the multifluid volume fraction advection scheme that updates 10 of the 16 variables in each cell is already very computationally intensive. Use of 32-bit arithmetic seems the best solution; it increases the performance to 840 Mflop/s and reduces the fraction of the time spent rearranging data in memory to only 24%. The reason for the unusually high fractions of the 1-D strip update performance that are achieved using 32-bit arithmetic on the Itanium-2 processor is that on this CPU 32-bit arithmetic runs at essentially the same speed as 64-bit arithmetic. For this CPU it may therefore be possible at essentially no extra cost to store the data in main memory in 32-bit form, thus effectively doubling the main memory capacity and bandwidth, but to perform all the 1-D grid strip updates in 64-bit form for whatever accuracy benefit this might yield.

32-bit 1.7 GHz PM	N = 32	N = 64	N = 128	64-bit 1.7 GHz PM	N = 32	N = 64	N = 128
Flops/Cell	8781	6206	5232	Flops/Cell	8787	6207	5145
Sec/Δt	0.437	2.23	14.0	Sec/Δt	0.716	4.00	26.4
Sec/Δt @N=512^3	1792	1142	893	Sec/Δt @N=512^3	2933	2046	1691
Mflop/s	658	730	786	Mflop/s	402	407	408
Fraction 1D Perf.	72%	67%	66%	Fraction 1D Perf.	72%	68%	66%

Performance of the 3-D multifluid PPM code with 32- and 64-bit arithmetic on an Intel Pentium-M CPU running at 1.7 GHz in a Dell Inspiron 8600.

32-bit 3.2 GHz P4	N = 32	N = 64	N = 128	64-bit 3.2 GHz P4	N = 32	N = 64	N = 128
Flops/Cell	8781	6206	5232	Flops/Cell	8787	6207	5145
Sec/Δt	0.325	1.64	10.3	Sec/Δt	0.421	2.32	15.4
Sec/Δt @N=512^3	1330	839	657	Sec/Δt @N=512^3	1724	1188	983
Mflop/s	886	993	1069	Mflop/s	684	701	703
Fraction 1D Perf.	74%	72%	70%	Fraction 1D Perf.	74%	72%	70%

Performance of the 3-D multifluid PPM code with 32- and 64-bit arithmetic on an Intel Pentium-4 CPU (extreme edition) running at 3.2 GHz in a Dell desktop PC.

32-bit 1.2 GHz It2	N = 32	N = 64	N = 128	64-bit 1.2 GHz It2	N = 32	N = 64	N = 128
Flops/Cell	8781	6207	5196	Flops/Cell	8787	6207	5109
Sec/Δt	0.350	1.95	13.0	Sec/Δt	0.447	2.52	17.4
Sec/Δt @N=512^3	1432	999	830	Sec/Δt @N=512^3	1833	1291	1115
Mflop/s	823	834	840	Mflop/s	643	646	615
Fraction 1D Perf.	87%	82%	76%	Fraction 1D Perf.	65%	62%	58%

Performance of the 3-D multifluid PPM code with 32- and 64-bit arithmetic on an Intel Itanium-2 CPU running at 1.3 GHz in the Unisys ES-7000. These results were obtained using the Intel 7.1 Fortran compiler, with the flag /O3 for 32-bit arithmetic and /fast for 64-bit arithmetic, and with the use of 8×8 grid pencils.

Parallel Computation

Although all the machines tested here support computation on a single CPU very well, with extremely well integrated graphics, only small 2-D problems can be run interactively in this mode. A huge advantage of the Unisys ES-7000 is its ability to support parallel computation on 16 CPUs with a shared memory programming model. This makes possible truly interactive 2-D computation, even at very respectable resolution. Together with Unisys, we demonstrated this point in August, 2003, by running complex flow problems on demand in just a few minutes, displaying the evolving simulated flows at the Wonders of Technology exhibit at the Minnesota State Fair. It should be clear from the performance figures quoted earlier that if 16 CPUs are used to update a 4×4 set of grid tiles, each with 256×192 cells, then we can expect a time step to take no more than about a seventh of a second for our ¾-million-cell grid. We can resolve a great deal of detail on a grid of this size (cf. www.lcse.umn.edu/StateFair-DemoCode). We need about 2000 time steps to do a run at this resolution, which, at our estimate of 7 steps per second, would take about 5 minutes. This strategy would work well for simulations in which nearly identical amounts of computational labor is expended in each grid cell. In this case each CPU would complete its grid tile update at almost the same instant. Then all could pause to exchange ghost cell data using the very low latency and high bandwidth of the shared memory system in the ES-7000. This data exchange would take very little time, after which the grid tiles could all be simultaneously updated again.

A more general parallel computing model exploits the shared memory of the ES-7000 to dynamically balance uneven computational loads among the several grid tiles. In this model, which we call SHMOD (see www.lcse.umn.edu/shmod), we decompose the problem domain more finely. We could, for example, use an 8×8 matrix of grid tiles, each with only 128×96 cells. According to the measurements in our tables, this finer domain decomposition would introduce an additional 19% to the cost for the run in a case with precisely the same amount of work to do in each grid tile. However, for dynamically varying loads coming from such physical processes as the movement of shocks and multifluid mixing layers, our overall computation could become very much more efficient. We can exploit the shared memory to allow any processor that becomes available for work to update the next grid tile in a carefully ordered sequence. Then, with our 4 tiles per processor, we would be able to let processors who had been given lighter loads proceed on to update tiles later in the sequence while processors given more difficult tiles finish up this greater work. If we are willing to guess at the value to use for the next time step based upon past history and incomplete data about the present step, then we can also let processors finishing work for this time step early proceed on to the next time step. If this strategy occasionally fails us, we can have the processors go back and repeat some of their work. The work they would repeat in such an instance is not wasted, since a more conservative strategy would have kept them idle otherwise.

The parallel computing strategy just described is very flexible, and it can be used on a cluster network or on a shared memory machine. On the cluster, the shared grid tile and ghost cell data must be passed around the cluster network by the application explicitly, with the locations of all this data managed by the application as well. This programming labor is unnecessary on a true shared memory machine like the ES-7000, since the grid tile data can simply be stored in arrays accessible transparently to all processors. If one is writing such a parallel application from scratch, it is obviously far simpler to do on a shared memory machine. However, even if one has gone to all the trouble, as our team has with our SHMOD framework, to implement this shared memory computing paradigm on clusters, there is still a huge advantage to a true shared memory machine. The reason for this is that the proprietary internal memory interconnection network has lower latency and higher bandwidth than any third-party cluster network. For very large problem sizes, where we have large chunks of data to transmit and seconds in which to do this, the latency of a cluster network becomes largely irrelevant. If the grid tiles or grid bricks are sufficiently large, and enough ghost cell data is supplied, we can update them for enough time steps so that the bandwidth of a third-party interconnect is sufficient to the task for our PPM codes.

However, for small problem sizes latency becomes very important. If the computation is to be efficient, small problems do not allow large enough grid tiles or grid bricks with enough ghost cell data to permit more than a single time step update. For the small transmissions involved, the bandwidth actually delivered also becomes much less than the link specifications might suggest. The result is that clusters are not good at running small problems fast, while shared memory machines excel at this very important job. We can see this from our earlier example of a dynamically load balanced simulation on a ¾-million-cell grid using 16 CPUs. If these 16 CPUs are in a shared memory machine like the ES-7000, we can get the run done interactively in just 5 minutes. On a cluster with a Gigabit Ethernet interconnect and with the simplest programming approach, in which the communication is not overlapped with computation, the run would take about 10 minutes. To make the run more efficient on the cluster, we could make each grid tile 4 times larger, or 512×384 grid cells, and update each tile for 4 rather than just 2 time steps. The fraction of redundant labor in ghost cell regions would then be unchanged, but there would now be four times as much data to communicate while the tile updates would take 8 times longer. Rather than spending half of the time sending and receiving data, we would now spend only a third of the time. But the total run would now take about an hour, and the efficiency would still be rather poor. One can see that clusters can do parallel computations efficiently only when the problems are very large. Shared memory machines, like the Unisys ES-7000, with their high bandwidth and low latency interconnects are efficient for problems whose solution we can see unfold during a time we are willing to wait.

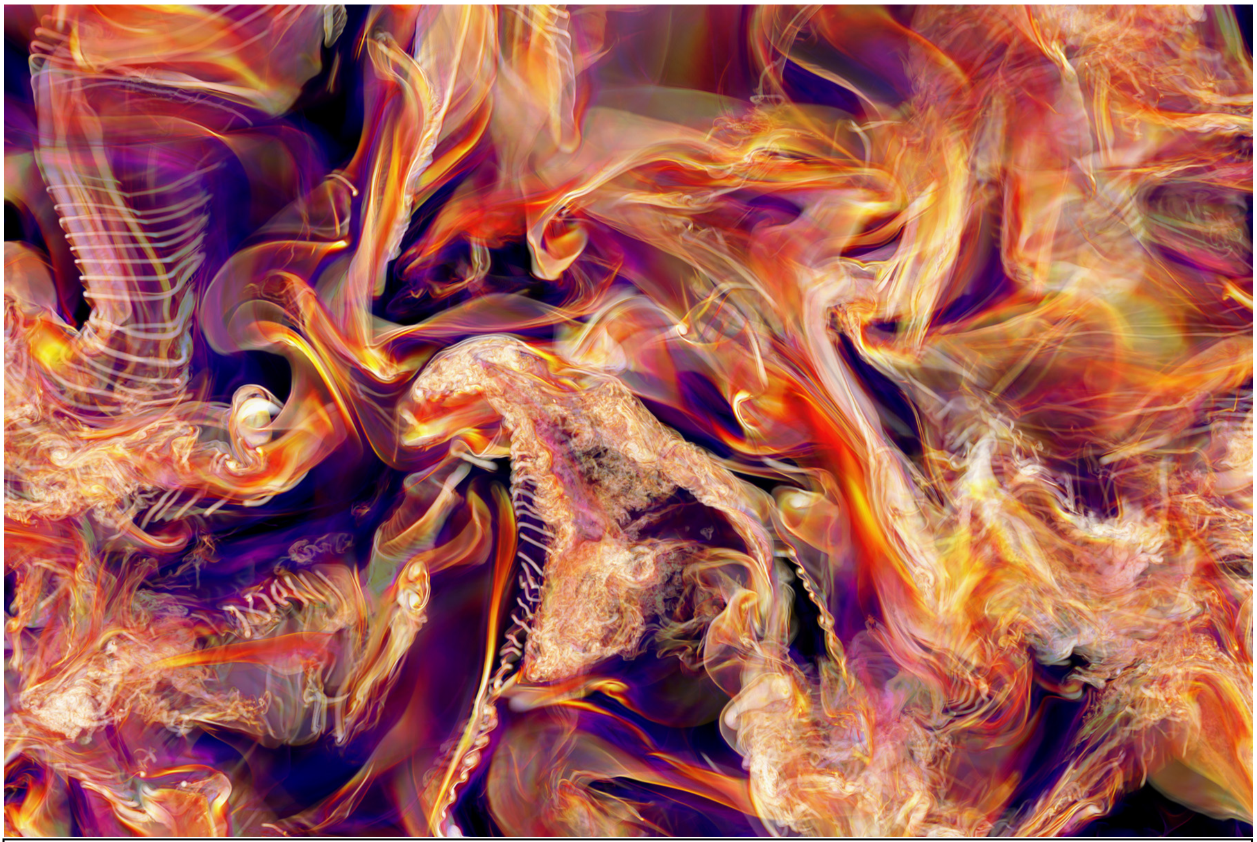
3-D Full Problem Performance

We here report results for computing a problem whose grid consists of a $2 \times 4 \times 8$ arrangement of grid bricks. We used our standard PPM code on the ES-7000, first employing grid bricks of 64^3 cells and then using bricks of 128^3 cells. We chose a problem similar to that used for the earlier tests, so that the performance levels can be directly and meaningfully compared. We also report results for computing a problem with the multifluid PPM code on the ES-7000 with a grids of $2 \times 4 \times 8$ grid bricks of 64^3 and of 128^3 cells. These parallel computations are done in our SHMOD framework, but with the brick records stored in the shared memory rather than on disk. The data transfers were carried out by extra threads of execution which copied data into our out of sections of memory-mapped files. The memory server threads all ran on a single CPU, and for the cases with 15 computational threads, each pinned to a particular CPU, the memory servers consumed about half of the 16th CPU. Using a first-touch mechanism, we were careful to have each computational thread lay out its workspace on memory within its own 4-CPU "pod" on the ES-7000. The brick data records that were updated were evenly distributed among the 4 pods in the machine. By pinning the computational threads to particular CPUs and by distributing the data upon which all these CPUs operate over the course of the computation, we were able to obtain excellent scaling of performance as the number of CPUs was increased. The multifluid problems involving 128^3 bricks consumed a great deal of the ES-7000's 32 GB memory while delivering 9¼ Gflop/s sustained performance on 15 CPUs. For these larger bricks, the standard PPM code delivered 11.1 Gflop/s. This is about 13% of the machine's 83.2 Gflop/s aggregate peak performance (for all 16 CPUs). We are very pleased with this result, especially since this performance is delivered to a real, very practical calculation, and it includes all costs of image data generation so that the results of the run can be studied. There is even about half a processor remaining unused, allowing the machine to remain responsive to such other tasks as database queries or dispatching generated image data over the network to a user. The difference in the performance of the standard PPM and of the multifluid version results mainly from the 3 times greater data bandwidth required by the somewhat slower, multifluid code.

32-bit 1.2 GHz It2 Standard PPM	Sec / Δt (2×4×8 64^3 bricks)	Gflop/s	Mflop/s per CPU	32-bit 1.2 GHz It2 PPMMF	Sec / Δt (2×4×8 64^3 bricks)	Gflop/s	Mflop/s per CPU
1 CPU	?	0.772	772	1 CPU		0.645	644
2 CPUs	?	1.538	769	2 CPUs		1.280	640
4 CPUs	?	3.064	766	4 CPUs		2.540	635
8 CPUs	?	6.064	758	8 CPUs		5.008	626
12 CPUs	?	8.988	749	12 CPUs		7.404	617
15 CPUs	?	11.115	741	15 CPUs		9.090	606

The performance measurements in the above table show a loss of only 31 out of 772 Mflop/s per CPU, or about 4%, in going from 1 to 15 CPUs working cooperatively to update these 64 grid bricks under our SHMOD framework. The loss in per-CPU performance is somewhat greater, about 6%, for the multifluid application, since it involves much greater data movement within the machine. For the larger bricks, the parallel performance losses are smaller. Only 2.2% in per-CPU performance is lost in going from 4 to 15 CPUs for these large bricks, while for the smaller bricks the corresponding loss is 3.3%. About 7.4% of the performance for a single CPU is lost here from the measurements of the previous section, because this application must read and write entire grid bricks from the main memory into the CPU task's private main memory, shuffling brick interiors, faces, edges, and corners in order to compose a complete grid brick augmented with the proper ghost cell regions.

32-bit 1.2 GHz It2 Standard PPM	Sec / Δt (2×4×8 128^3 bricks)	Gflop/s	Mflop/s per CPU	32-bit 1.2 GHz It2 PPMMF	Sec / Δt (2×4×8 128^3 bricks)	Gflop/s	Mflop/s per CPU
4 CPUs	?	3.151	788	4 CPUs		2.600	0.650
8 CPUs	?	6.263	783	8 CPUs		5.112	0.639
12 CPUs	?	9.332	778	12 CPUs		7.548	0.629
15 CPUs	?	11.559	771	16 CPUs		9.255	0.617

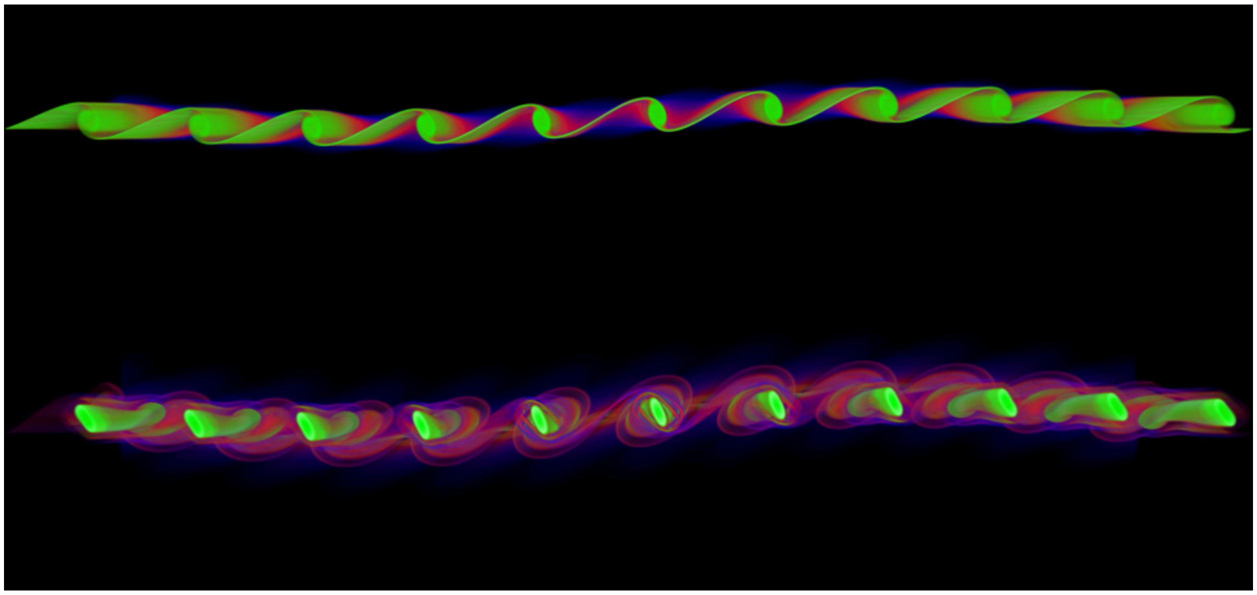


The distribution of the logarithm of the magnitude of vorticity is shown in a thin slice of the 2048^3 simulation volume of a PPM simulation of Mach 1 decaying homogeneous turbulence. The flow is shown in the midst of the transition to fully developed turbulence (t = 1.05). White regions have the largest vorticity, and as the vorticity decreases the color changes from white to yellow, red, purple, blue, and finally for very weak vorticity values the flow is made transparent. Shear layers breaking up into lattices of roughly parallel vortex tubes can be seen as well as the results of such structures developing 3-D instabilities leading to local turbulence.

Applications

For several years we have been studying the behavior of compressible, homogeneous turbulence. From simulations carried out at various national centers over this period, we have developed the conjecture that the rate at which energy is transferred from the local large-scale flow to small-scale turbulent motions is related to the flow topology. We have identified a correlation of this energy transfer with regions in which the flow is compressing in one dimension and expanding in the other two – the type of flow that results in air when you claps your hands. In contrast, we identified a correlation of inverse energy transfer, from small scales to large, in regions where the flow is compressing in two dimensions and expanding in the other one. This type of flow results when you squeeze a tube of toothpaste. An example in air is a tornado. These unexpected correlations emerge from our simulations of very complicated flows such as the homogeneous turbulence simulation shown in the figure above.

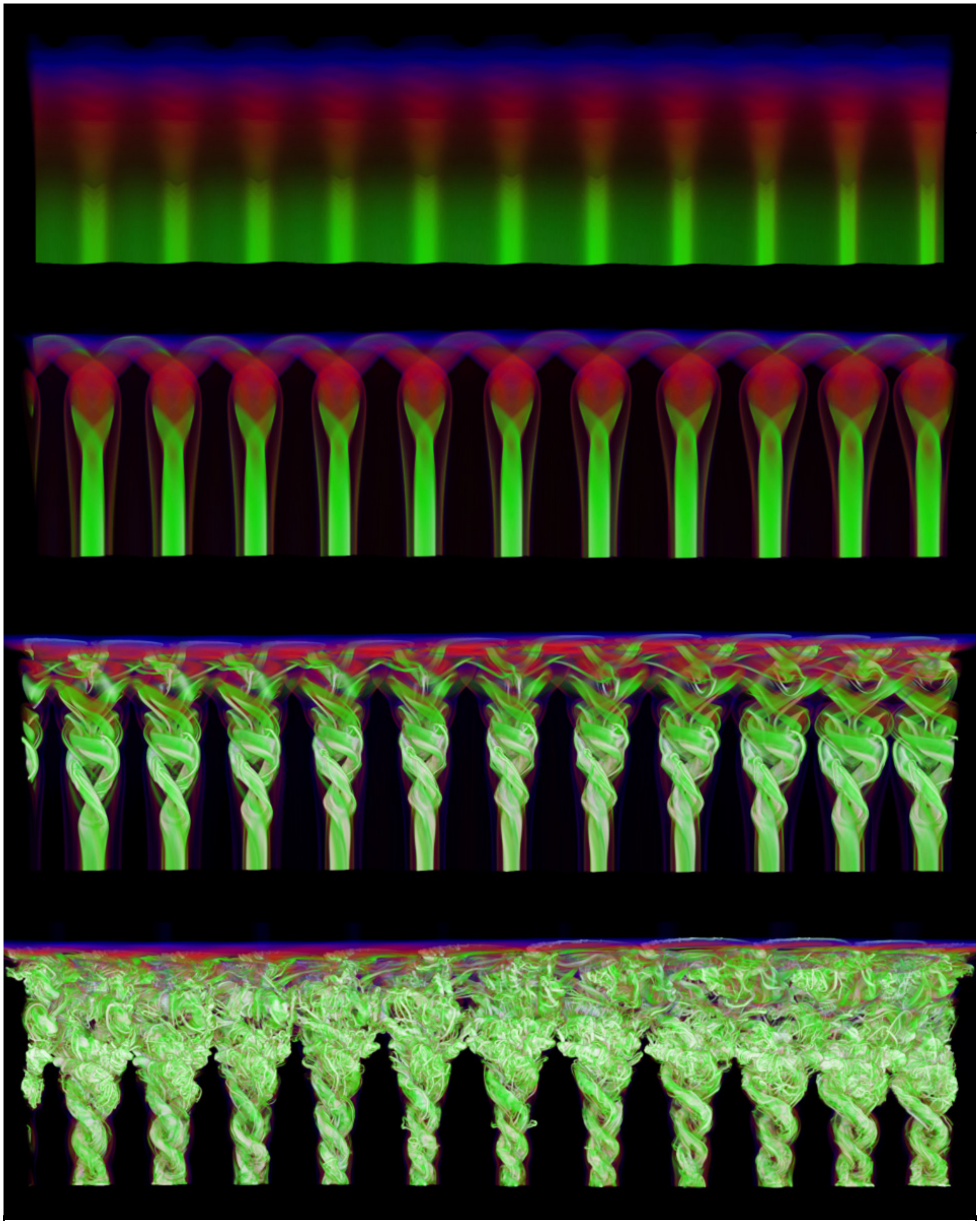
In order to clarify the role of the flow topology, which can be measured quntitatively by the determinant of the rate of strain tensor, we constructed last year a set of carefully designed experiments. We computed these flows on grids of up to a billion cells using our PPM code on the Unisys ES-7000. The textbook example of shear instability, the Kelvin-Helmholtz instability in 2D, is neither conducive nor destructive to turbulent energy, according to our model for the energy transfer. First, we point out that in the absence of a process such as gravitational separation of two fluids and wind forces acting on



The distribution of the logarithm of the magnitude of vorticity is shown in a PPM simulation of a Mach 1 shear layer with two sinusoidal perturbations of wavelengths differing by a factor of 11. A Mach 1 circulating flow above and below the shear layer squeezes the side of the layer near the viewpoint and pulls the layer apart on the far side. The layer is viewed at two early times in the development of the Kelvin-Helmholtz instability.

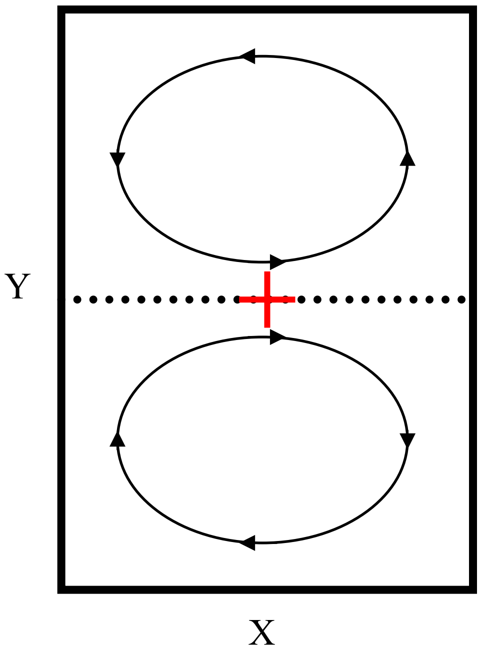
fluids with very different inertia, it is hard to understand how the shear layer that forms the initial condition for the Kelvin-Helmholtz instability can have come to be. To create such shear within a single fluid in the absence of gravity, it is likely that the fluid was compressed in the dimension perpendicular to the shear layer. The passage of a three-shock intersection line is another mechanism from which such a shear layer might have formed, but that would require supersonic flow. So it is unlikely to find a shear layer ready to undergo the Kelvin-Helmholtz behavior that has not already experienced a local compression in one dimension with expansion in the other two. Nevertheless, regardless of how the shear layer formed, a strictly 2-D layer cannot generate turbulence as we know it in 3-D, since the evolution of the layer is to accumulate vorticity, originally in an extremely thin layer, into ever larger vortex cores. Instability in the third dimension, resulting from the interaction of these vortex tubes, is required to generate turbulence.

We set up a classic 2-D planar shear layer and perturbed it with two sinusoidal modes of very different wavelength. In 2-D this initial set up would generate the familiar Kelvin-Helholtz behavior. We then added an incompressible circulation in two counter-rotating rolls, one above and one below the layer. This was done in such a way that this incompressible flow squeezed down on the shear layer on one side (in the third dimension) and pulled it apart on the other. Our model for the energy transfer to turbulence indicates that on the side of the layer that is squeezed, energy should be transferred to turbulent motions on the smallest possible scales, while on the other side of the layer, any existing turbulence should be reduced in intensity. This experiment provides a fairly clear, yes-or-no answer to the question of whether our correlations make sense. The result can be viewed in a movie on our Web site at www.lcse.umn.edu/-AVIs. It is not as clear an answer as we had hoped, but it is very illuminating. We initialized the shear layer with a width of a few cells and a hyperbolic tangent velocity profile. The squeezing flow intensified this shear, enhancing the unstable Kelvin-Helmholtz behavior on this side of the layer in the third dimension. Where the vertically expanding flow was pulling the layer apart, unstable growth was initially much slower. The initial development of the flow is not turbulent. Instead highly organized, parallel vortex tubes form on the side of the layer that is being squeezed. On this side of the layer, these vortex tubes rotate more rapidly than the fatter vortex tube sections that form on the other side, where the shear layer is being pulled apart. The faster rotation on the squeezed ends of these vortices causes them to split into two,



Here the shear layer of the previous figure is viewed from above, so that the different behaviors, described in the text, between the side of the layer that is squeezed by a Mach 1 circulating flow and the side that is pulled apart by it can clearly be seen. On the side that is squeezed, vortex tubes form and intensify much more rapidly. As the circulating flow carries these vortex tubes to the expanding side of the layer, 3-D instabilities generate turbulence, which is subsequently diminished by the vertically expanding flow.

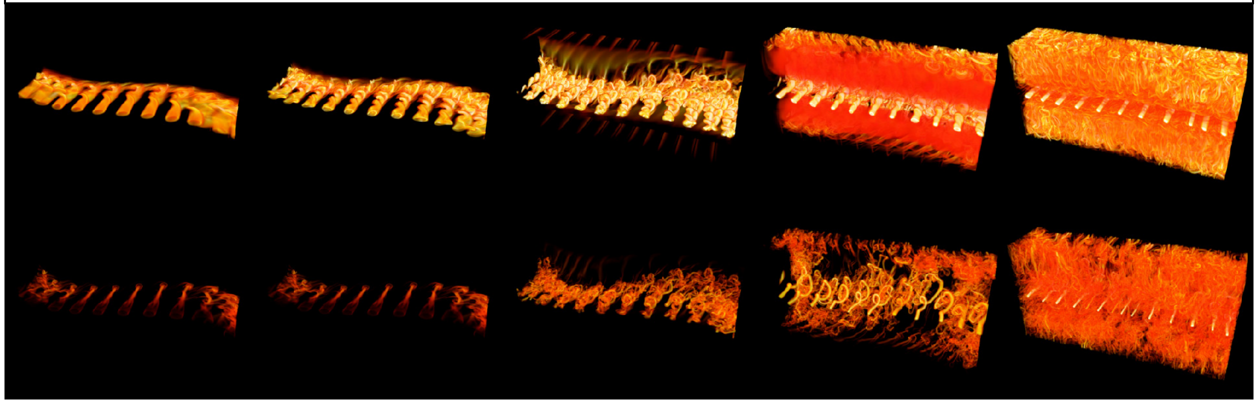
which then intertwine like a braided rope. The circulating flow stretches these vortices on the ends that are being squeezed, so that they remain roughly parallel, while the compression of the vortices along their lengths on the other end of the layer is destabilizing, and turbulence develops there.

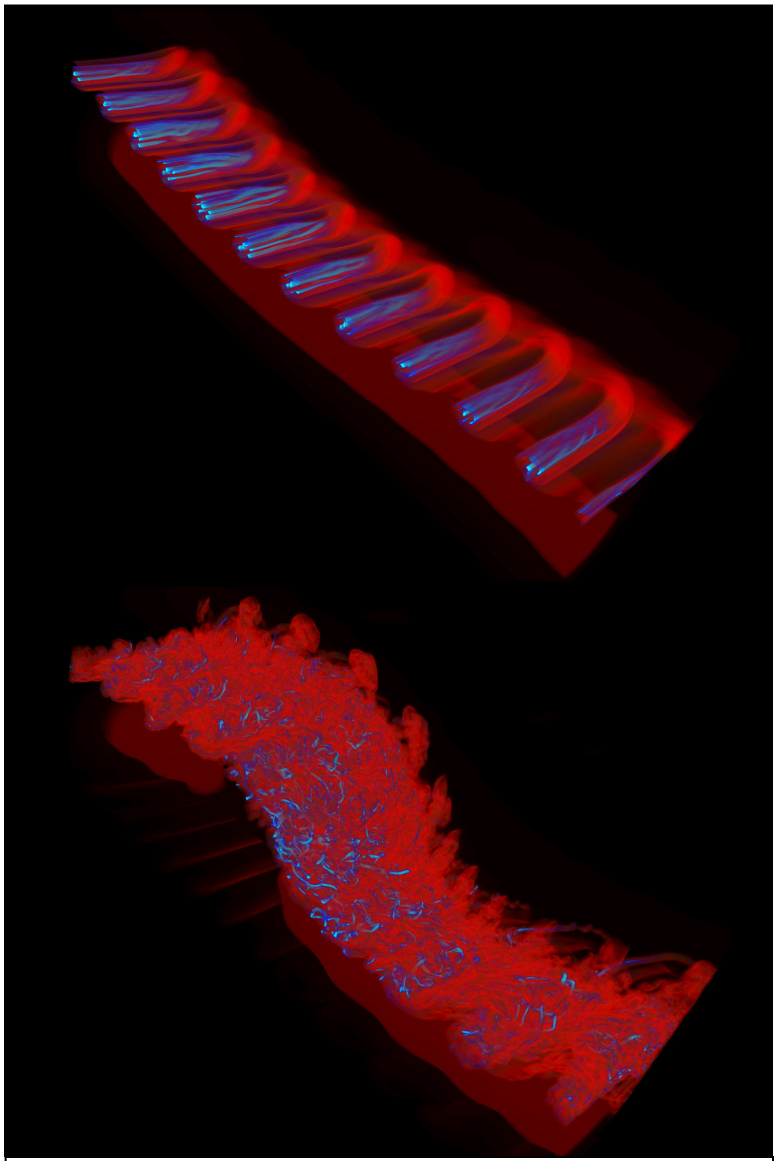


The circulating flow in a plane perpendicular to that of the initial shear layer (dotted line) is indicated here. In this view, the gas in the lower half-plane flows into the paper and the gas in the upper half plane flows out of the paper.

At first glance, the behavior described above may cause one to think our model for the turbulent energy transfer is wrong. However, one must analyze this flow in the co-moving frame, in which our model is intended to apply. The conditions that set up the growth of the small-scale, intense vortex tubes exist in the region of the flow where shear is being intensified by squeezing upon it. As this region of the flow circulates into the side of the shear layer that is being expanded, turbulence develops and is subsequently advected into the surrounding flow and diminished in intensity by the circulating flows above and below the shear layer. This behavior is entirely consistent with our model, both phenomenologically and quantitatively. In fact, the statistical data is fit beautifully using the same dimensionless coefficients of the terms in our model that we derived from the homogeneous turbulence data. This experiment serves to clarify why the transfer of energy to turbulence depends upon the local flow topology. This topology does not generate turbulence directly, but it generates the intensified shear that in turn creates vortex tubes through the action of the Kelvin-Helmholtz instability, and additionally it stretches these vortices with the result that they are greatly intensified. The vortex-vortex interactions that actually produce turbulence are a subsequent phenomenon. However, it is very important to recognize that this flow topology provides a context in which these 3-D vortex-vortex interactions can play out, since the alignment of the individual vortices within the plane of the shear is not constrained.

A matrix of turbulent shear layer simulations performed using the PPM code on the Unisys ES-7000. Simulations in the upper row have all velocities 4 times larger in magnitude than the corresponding simulations in the lower row. For the top center simulation, the flow parallel to the shear layer is ±Mach 1, and the peak magnitude of the circulating flow is Mach 1 also. As we move along the top row of this figure, the shear velocity remains Mach 2 (relative velocity), while the speed of the circulating flow decreases or increases by one factor of 2 for each simulation that we go to the left or to the right from the central one. These volume renderings show the logarithm of the magnitude of the vorticity. Grids of 128×256×512 cells were used for all these runs.

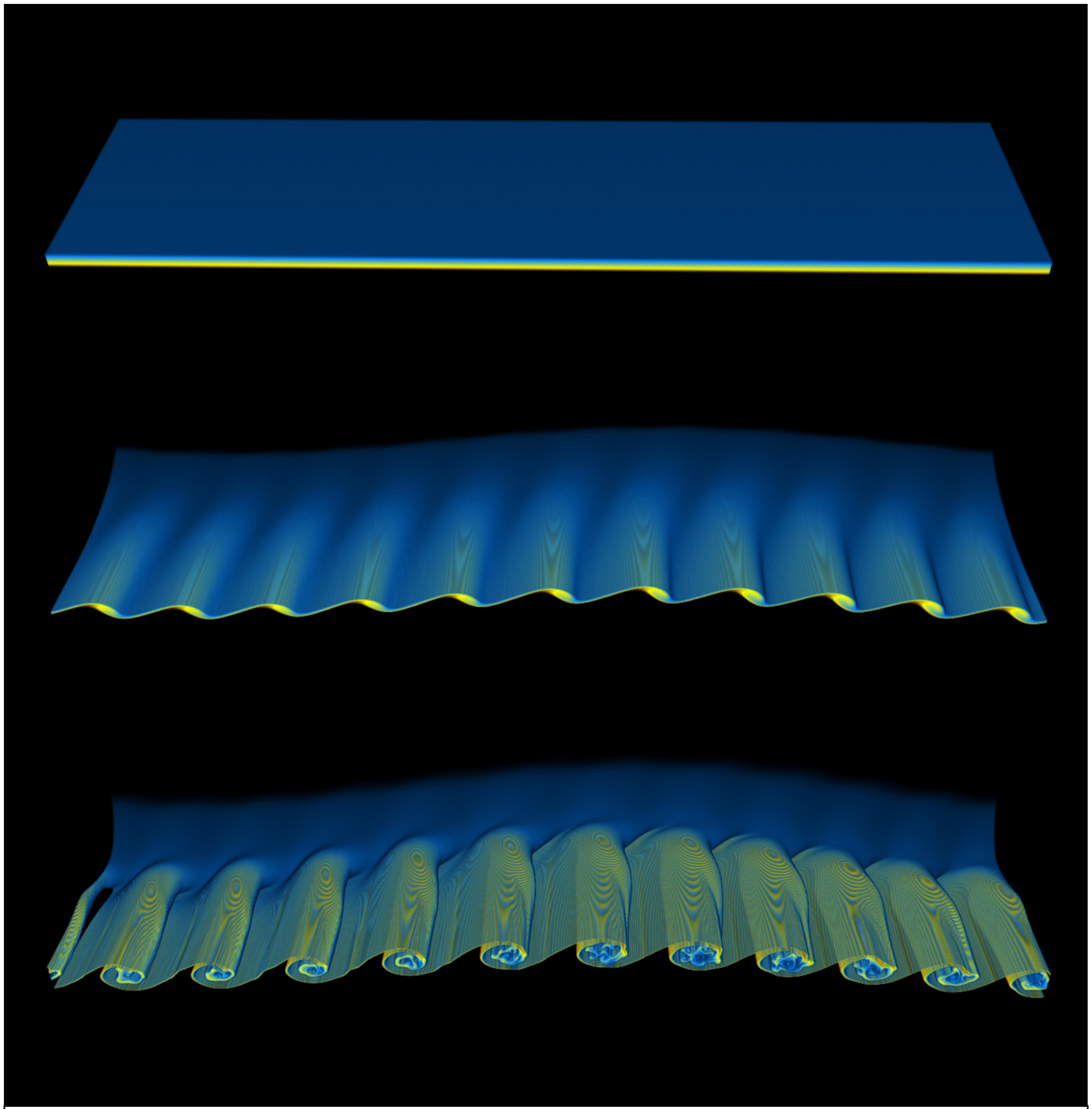



Vorticity structures in the multifluid shear layer instability problem described in the text. The shear layer is viewed from the sulfur hexafluoride side, with the near edge experiencing squeezing from the circulating flows in the two fluids. The vorticity associated with this circulating flow in the air on the other side of the shear layer is just visible in these visualizations. The behavior in this multifluid problem is quite similar to the single-fluid behavior discussed earlier.

Our team has been extending this work recently to two-fluid problems, using the new multifluid PPM code whose performance we have reported here in detail. We first used the Unisys ES-7000 to validate this code in 2D by simulating shock tube experiments performed at the University of Arizona to study the Richtmyer-Meshkov instability of a shock-accelerated interface between SF_6 and air. That work will be reported in another document. We are now using the ES-7000 to run the multifluid PPM code on 3-D shear layer instability problems like the one just presented, but with the shear along an initially planar interface between SF_6 and air. We will use the data from these runs to test our ideas on turbulent energy transfer in situations involving two-fluid mixing. This investigation is of considerable interest to our DoE sponsors. We began this work by first performing a parameter study of these shear layer problems on the ES-7000. Some results of that study are shown at the bottom of the previous page.

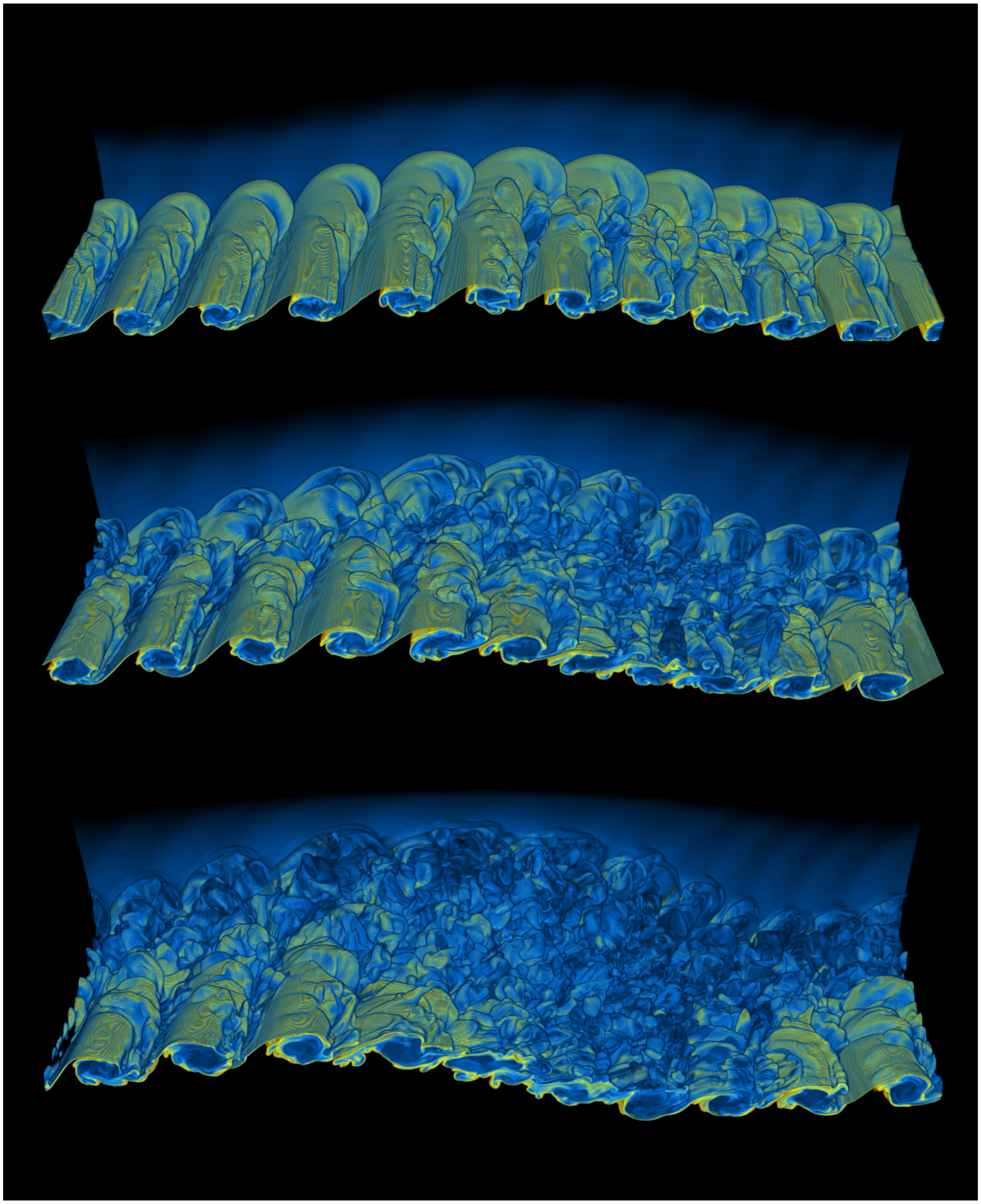
We are now in the process of performing a multifluid simulation on a billion-cell grid, but results from a run on a grid of 256×512×1024 cells are shown below. Here we represent both fluids as gamma-law gases, where gamma for the air is 1.4 and for the sulfur hexafluoride is 1.09. We take the SF_6 to be 4.88 times denser than the air when they are at the same pressure and temperature. The relative shear velocity between the air and the SF_6 is 0.399 times the sound speed in the air, or just the sound speed in the SF_6. The amplitudes of the circulating velocities in the air and in the SF_6 are, respectively, Mach 0.2 and Mach 0.226. Finally, the initial amplitudes of the fundamental and 11th harmonic sinusoidal vertical velocity disturbances are Mach 0.05 and 0.0045 relative to the speed of sound in the air.

The vorticity structures in this developing, multifluid shear layer are shown above, looking at the layer from the sulfur hexafluoride side. The development of the vorticity structures is not too different from the earlier, single-fluid case, despite the density contrast of 4.88 in this problem. Here the Mach numbers are about a third as great as in the earlier case also. We are interested in this problem not only in

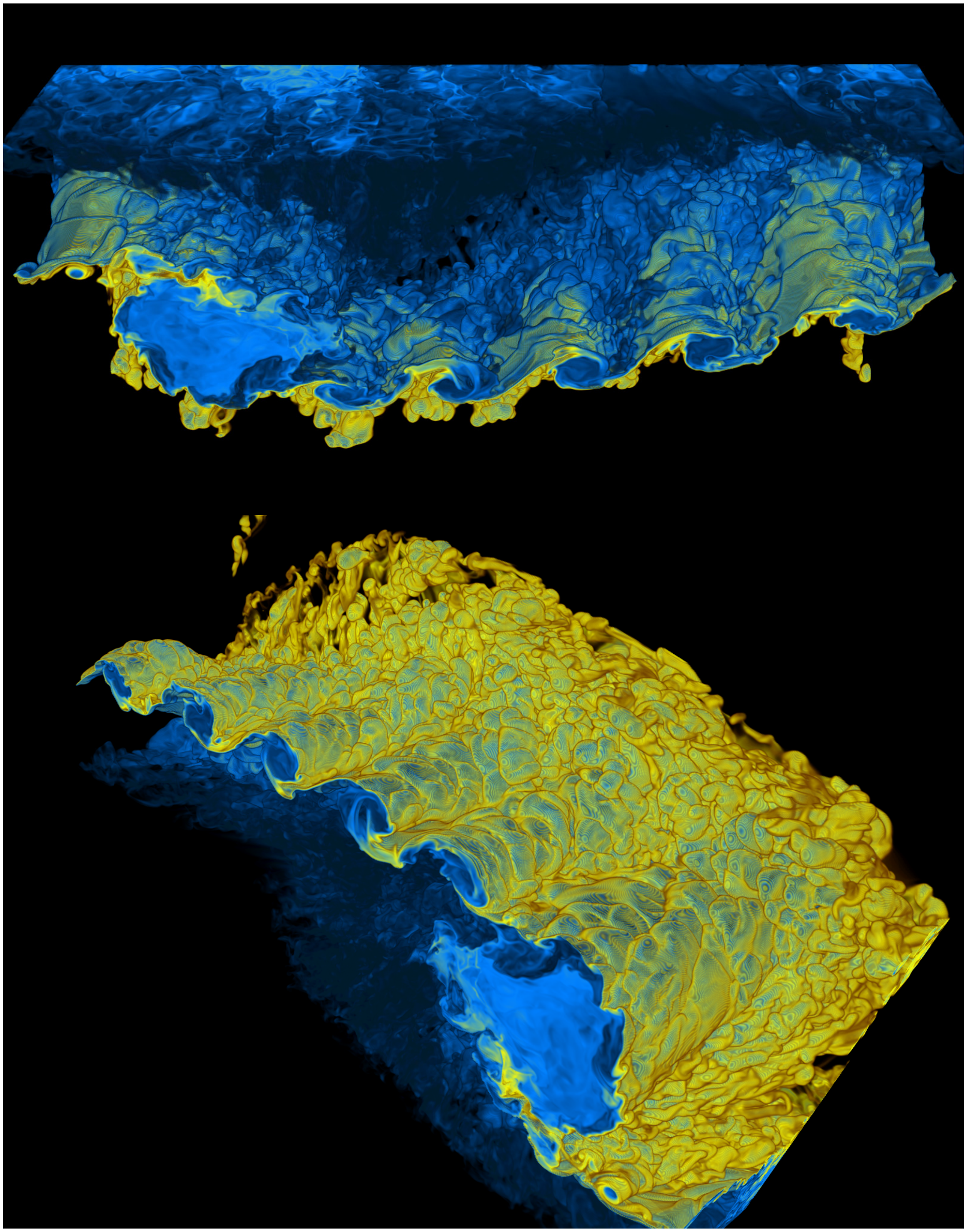


The developing multifluid shear layer is seen from above, looking through the air and into the sulfur hexa-fluoride below. The fraction of SF_6 that is mixed with the air is visualized, with regions of either pure fluid rendered transparent. Blue and dark blue represent mixes with more air and even more air, while yellow and brown represent mixtures of more and still more SF_6. The moiré patterns are rendering artifacts caused when the transition takes place over distances comparable to just one or two grid cell widths. The initial transition layer for both the shear and the fluid fractional volume was 16 grid cells thick in this simulation. The mixed fluid present initially is rapidly transported into the 11 small breaking waves, and the mixing layer elsewhere is then given the thinnest representation possible with this grid and this numerical scheme. Note that at the far edge of the shear layer, where it hits a reflecting wall, the circulating flow entrains mixed fluid that is mostly air.

the development of turbulence, and the extent to which our model can predict where that occurs, but we are also interested in the mixing of the two distinct fluids by this developing turbulence. That process is best visualized by looking at the multifluid mixing fraction, which is very carefully tracked by this new multifluid PPM code.



The multifluid shear layer, like the single-fluid one, develops vorticity most strongly at the near edge, where the circulating flow is squeezing on it. Again, the development of the turbulence occurs downstream, which in this case means near the center of the layer. Finally, at the far edge, the circulating flow draws the mixing fluids into the unmixed regions. Here the intensity of the turbulence is



diminishing, but the mixing of the fluids is no longer confined to a thin layer. The entrainment of the mixed gases into the pure sulfur hexafluoride is less rapid, as the final views from both the air side (top) and SF_6 side (bottom) of the layer attest.

Summary

The performance measurements reported here demonstrate that with appropriate coding effort excellent computational performance can be obtained with the latest CPUs from Intel. These CPUs support vector computation from arrays located in their cache memories. Although the vector performance is lower as a fraction of the processor's theoretical peak performance than it was for a Cray vector CPU of the distant past, it is nevertheless very good, especially for the price. The performance enhancement over good scalar code is roughly a factor of 2.5 for the Itanium-2 CPU and a factor of 1.7 for the Pentium-4, with both using 64-bit arithmetic. With 32-bit arithmetic, the vector-to-scalar ratio remains 2.5 for the Itanium-2 CPU but increases to 3.2 for the Pentium-4. The quite comprehensive tests of processor performance embodied in the 2-D grid tile and 3-D grid brick updates, with image generation, for the various PPM codes reveal that the limitations of the supporting memory systems for these powerful CPUs reduce the delivered performance substantially when they are using 64-bit arithmetic. Keeping this performance degradation due to the memory system under control requires considerable care in algorithm and code design. All effective strategies benefit from larger cache memories than are now found on these CPUs. The parallel performance measurements for 16 CPUs in the Unisys ES-7000 show very good scaling for such a shared memory machine. These measurements highlight the machine's ability to deliver excellent performance even on small problems which, in the case of 2D flows, can be delivered interactively in just minutes rather than hours or days.

Acknowledgement

This work was supported by the University of Minnesota's Digital Technology Center through the Minnesota Supercomputing Institute and also by the NSF through the LCSE's partnership with NCSA under the PACI program. Work in developing the numerical algorithms used in these tests has been supported largely through the DoE Office of Science MICS and NNSA ASCI programs. It is a pleasure to acknowledge Unisys Corporation's generous donation of the 32-processor ES-7000 machine at the LCSE that was used in this work.