A Study of Multithreaded Benchmarks on the Hewlett-Packard X- and V-Class Architectures

Sharon Brunett

California Institute of Technology
Center for Advanced Computing Research
Mail Code 158-79, Pasadena CA 91016, USA
email: sharon@cacr.caltech.edu
phone: (+1 626) 395 6734, *fax:* (+1 626) 584 5917

Abstract

The Hewlett-Packard X- and V-Class ccNUMA systems appear well suited to exploiting coarse and fine-grained parallelism, using multithreading techniques. This paper briefly summarizes the multilevel memory subsystem for the X- and V-Class platforms. Typical MPP distributed memory programming concerns for the codes under investigation, such as explicit memory localization and load balancing, are compared to relevant issues when porting and tuning for the X- and V-Class.

This paper uses two small benchmarks as the basis for investigating differences running multithreaded codes in SPP-UX and HP-UX environments. One code is from the Command, Control, Communication and Intelligence (C3I) Parallel Benchmark suite, shown to have the potential for large-scale parallelization with straightforward multithreading techniques. The second benchmark exhibits the computationally dynamic behavior of a thermally-driven explosion model. Both codes are shown to stress the HP systems' ability to keep memory close to processors and appropriate threads of execution.

1 System Architecture High-Level Overview

The Hewlett Packard X-Class and V-Class servers are symmetric multiprocessor (SMP) cache coherent nonuniform memory-access (ccNUMA) systems, providing the illusion of simple, integrated memory. The fundamental building block of the X and V-Class systems is the hypernode. Each hypernode is an SMP, containing multiple processors, a crossbar, caches and synchronous DRAM (SDRAM) distributed across multiple memory boards. Hypernodes are connected through a ring interface, referred to as Coherent Toroidal Interconnect (CTI).

1.1 Globally Shared Memory

All processors in X and V-Class servers share memory within a hypernode (local memory) and across the entire collection of hypernodes (remote memory). The global shared memory (GSM) subsystem is two-level; each of which is tuned for a particular class of data sharing. Level one includes a crossbar, connecting memory, processors and I/O within a hypernode. The crossbar provides high bandwidth, low latency non-blocking access from processors and I/O channels to hypernode local memory. Level two encompasses the interconnection between hypernodes through the use of CTI rings. The CTI is a collection of rings used to access remote memory across hypernodes. CTI is specially designed to enable extremely high-bandwidth (15.3 Gigabytes/second) data movement between processors, I/O devices and memory on a node. Memory references not satisfied by level one memory subsystem requests use the crossbar and CTI (level two) interconnection, to access data not in hypernode local memory.

1.2 Globally Shared Memory and Cache Coherence

All processor references to memory, cause copies of the accessed data to be encached into either the instruction or data cache of each processor. Processor local cache holds the most frequently accessed data. When cache misses are encountered, an attempt is made to retrieve data from node local SDRAM. When

requested data is not resident in local cache nor local memory, the data resides in memory of another node. Such remote data is obtained via the CTI interconnect. Since remote memory accesses come with a high latency cost, each node has local memory dedicated to CTI cache. The CTI cache is responsible for caching data accessed by other nodes, thus reducing the time to retrieve remote memory.

When a processor modifies data within its data cache, and another processor references the same data, stale data conditions exist. The X- and V-Class hardware supports cache coherence, thus each processor is assured caches always contain the latest data values. Hardware supported cache coherence relieves the programmer from explicitly flushing cache and tending to expensive synchronization details. Maintaining coherent copies of processor cache is achieved by adherence to the following rules:

- Any number of read encachements on a cache line can occur concurrently. A cache line can be read-shared in multiple caches.
- A processor must exclusively "own" a cache line in order to write data into a cache line.
- Modified cache lines must be written back to memory from the cache before overwriting occurs.

Particular hardware characteristics for the platforms used in our benchmarking are listed in Table 1.

System	CPU	CPUs/Node	Nodes	Clock Speed	Data/Instr Cache	Memory/Node	OS
V2250	PA-8200	16	2	240 MHz	2MB/2MB	8GB	HP-UX 11.01
X2000	PA-8000	16	16	180 MHz	1MB/1MB	4GB	SPP-UX 5.3

Tab. 1: Platform Specifics

2 The Benchmark Problems

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite [1] consists of eight problems chosen to compactly represent the essential elements of real C3I applications. The C3I representative benchmark discussed in this paper is Threat Analysis—a time-stepped simulation involving the trajectories of incoming ballistic threats, with computation of options for intercepting the threats. The benchmark includes input data sets, a sequential C program, and output sets to validate correctness. Threat Analysis is computationally intensive, compact, and involves non-trivial data and control structures. It should make a good test for compiler parallelization effectiveness and multithreaded performance analysis on the X- and V-Class architectures.

3 Threat Analysis

The Threat Analysis problem is a time-stepped simulation of the trajectories of incoming ballistic threats with computation of options for intercepting the threats. The input to the problem consists of (i) the trajectories of a set of incoming threats, and (ii) the locations and capabilities corresponding to a set of weapons used to intercept incoming threats. For each threat and weapon pair, the program computes time intervals over which a given threat can be intercepted by a particular weapon. The benchmark provides five different input scenarios.

3.1 Sequential Threat Analysis

Program 1 gives a slightly simplified, high-level pseudocode representation of the algorithm used in the sequential Threat Analysis program.

The program computes a set of tuples of the form (threat, weapon, interval) indicating which threats can be intercepted by a given weapon over the time interval `t1...t2`. There can be zero, one, or more intervals associated with each (threat, weapon) pair. The `t1` and `t2` interception times within the inner loop are computed using time-stepped simulations of threat and weapon positions, and are not parallelization candidates.

The three nested loops in the program are not immediately parallelizable, because all iterations increment the `num_intervals` count and assign to the `intervals` array. The indices that a particular iteration assigns to cannot be determined without first executing the prior iterations. However, these shared variables are the

only obstacles to parallelization of the outer two loops, as computation of intervals for each (threat, weapon) pair are otherwise independent of each other.

```
ThreatAnalysis(
      in num_threats, in threats[],
      in num_weapons, in weapons[],
      out num_intervals, out intervals[])
{
      declare threat, weapon;
      declare t0, t1, t2;
      num_intervals = 0;
      for (threat = 0 ..  num_threats - 1)
          for (weapon = 0 ..  num_weapons - 1) {
              t0 = initial detection time of threat;
              while (weapon can intercept threat in [t0...impact time of threat]) {
                    t1 = first time after t0 that weapon can intercept threat;
                    t2 = last time after t1 that weapon can intercept threat;
                    intervals[num_intervals] = (threat, weapon, [t1 ..  t2]);
                    num_intervals = num_intervals + 1;
                    t0 = t2 + 1;
              }
          }
}
```

Program 1: Sequential Threat Analysis

3.2 Multithreaded Threat Analysis

It is relatively straightforward to manually modify the sequential Threat Analysis program into a multithreaded code. Program 2 gives a slightly simplified, high-level pseudocode representation of the algorithm used in the multithreaded Threat Analysis program.

The outer loop over all `threats` is replaced by a multithreaded loop in which each iteration is responsible for a different `chunk` (i.e., subrange) of the `threats`. The problem of the shared variables is solved by modifying the algorithm so that each iteration increments its own `num_intervals` count and assigns to its own section of the `intervals` array. Declarations of other variables are localized by moving them into the inner blocks. The outer-loop iterations are now completely independent of each other and can be executed by separate threads.

The drawback of this multithreaded implementation is that it requires a larger `interval` array than the sequential program. Since there is no way to determine in advance the number of intervals each iteration will compute, each iteration's section of the `interval` array must be generously oversized. Therefore, the larger the number of `chunks`, the larger the `interval` array. A distributed `interval` array approach would work well for an MPI implementation of this algorithm. Each processor would get a unique chunk of `threats` and updates a local portion of the `interval` array. A simple global operation would gather the distributed `interval` array into a single solution set.

Efficient parallelization of Threat Analysis requires more than sequential program loop parallelization. It involves significant modification of the underlying sequential algorithm to uncover the high-level purpose of the program, and develop an alternative approach to uncover data dependencies and addresses common compiler restrictions. Optimizing Threat Analysis, requires paying special attention to typical coding constructs, such as: tests within loops; ambiguous pointers; indirect memory references; subroutine calls; and loop invariants. Many of these same code tuning techniques are transferable to algorithms running in MPP environments. Although the data locality issues are, of course, different, eliminating clutter and permitting good compiler optimizations are, in general, helpful for improving runtime performance.

Table 2 shows execution time on the X- and V-Class architectures for the multithreaded Threat Analysis code, given various levels of optimizations. The compiler options `+O3` and `+Oparallel` turn on block-level, routine-level and file-level optimizations in addition to causing the compiler to recognize user-specified pragmas involving parallelism. Note that performance improved dramatically by compiling with `+Olibcalls` and `+FPD`. `+Olibcalls` uses millicode versions of frequently called trigonometric functions, in place of the

standard calls. +FPD avoids floating point traps on denorms. Higher clock speed and larger cache sizes of the V-Class are major contributors to faster execution times than measured on the X-Class.

```
ThreatAnalysis(
    in num_threats, in threats[],
    in num_weapons, in weapons[],
    in num_chunks, out num_intervals[], out intervals[][])
{
    declare chunk;
    #pragma multithreaded
    for (chunk = 0 ..  num_chunks - 1) {
        declare first_threat, last_threat;
        declare threat, weapon;
        declare t0, t1, t2;

        first_threat = (chunk*num_threats)/num_chunks;
        last_threat = ((chunk+1)*num_threats)/num_chunks - 1;
        num_intervals[chunk] = 0;
        for (threat = first_threat ..  last_threat)
            for (weapon = 0 ..  num_weapons - 1) {
                t0 = initial detection time of threat;
                while (weapon can intercept threat in [t0 ..  impact time of threat]) {
                      t1 = first time after t0 that weapon can intercept threat;
                      t2 = last time after t1 that weapon can intercept threat;
                      intervals[chunk][num_intervals[chunk]] = (threat, weapon, [t1 ..  t2]);
                      num_intervals[chunk] = num_intervals[chunk] + 1;
                      t0 = t2 + 1;
                }
            }
    }
}
```

Program 2: Multithreaded Threat Analysis

Respectable speedups occur with relatively small input data sets (1,000 threat sets) across a few processors. Good performance on larger numbers of processors is expected with larger scenario collections of weapons and threats, generating more independent parallel threads of execution.

CPUs	HP V2250 Timings Opt. A (sec)	HP V2250 Timings Opt. B (sec)	HP X2000 Timings Opt. A (sec)	HP X2000 Timings Opt. B (sec)
1	211	463	280	591
2	106	231	140	296
4	53	115	70	148
8	27	58	35	74
16	13	31	18	38
24	9	21	12	44
32	7	16	9	34

Tab. 2: Multithreaded Threat Analysis Performance. Optimizations A: Source Code Pragmas and +O3 +Oparallel +FPD +Olibcalls; Optimizations B: Source Code Pragmas and +O3 +Oparallel

4 Thermal Explosion Benchmark

The Thermal Explosion benchmark simulates an explosive wave initiating chemical reactions in reactive materials. The implementation is meant to stress a system's ability to keep memory close to processors. Benchmark execution across various spatial parameters will help assess system performance characteristics as parallelism is increased and caches are saturated.

The simulation begins with a detonation wave, composed of a lead shock wave, initiating a chemical reaction in a reactive material. In turn, chemical energy is released which sustains the shock wave. The chain-branching reaction mechanism is described at length in [2]. The mesh is kept static throughout the simulation, causing highly time-variable work-loads per grid point to move through the mesh along with the propagating explosive wave. As the explosive wave moves through the spatial extent, the system's ability to keep memory "close" to processors is increasingly important for large and complex problems adding to the work-load imbalance. In addition, sufficient numbers of concurrent threads must be available for good performance.

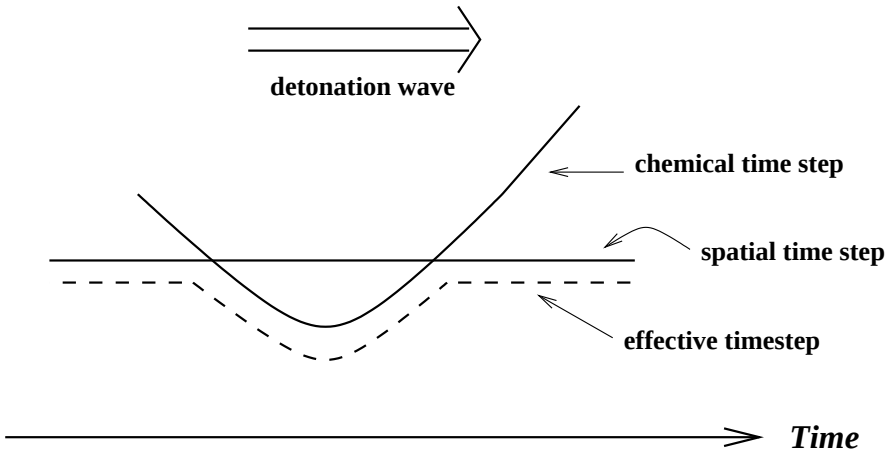
Distributed memory algorithms need to periodically re-distribute the computational work-load in order to keep all processors sufficiently busy. Not only does this task add communication overhead, bookkeeping and data movement preparation add to the cost of keeping processors uniformly busy with the simulation task at hand. For particularly dynamic work-loads, the overhead for load balancing can be unacceptable in relation to the total execution time. The Thermal Explosion code is implemented such that an effective timestep drives the imbalance of work at each grid point, as differential equations are solved.

The differential equations solved in the simulation consists of:

(i) A temperature field, with spatial diffusion and enthalpy generation from the reaction.

(ii) A set of chemical species concentrations, which responds to a Gruenesisen-type reaction network.

Solving these equations involves two different timesteps. The spatial diffusion solution involves a timestep proportional to the square of the mesh spacing. The reaction terms provide a chemical timestep derived from the maximum reaction rate driven by the temperature. During fierce reactions, very small timesteps are necessary for high fidelity simulation. The timesteps will be large for cold and fully-reacted conditions. Figure 1 depicts the work-load imbalance over time as the simulation evolves. Much more computation will be necessary at grid points with small timesteps, thus stressing the system architecture's ability to support multithreaded implementations that depend on efficient memory access.



$$\texttt{Imbalance} = \max \frac{\text{Spatial timestep}}{\text{Chemical timestep}}$$

Fig. 1: Workload Imbalance

The multithreaded Thermal Explosion implementation pays special attention to pushing parallelism to the outer level loops, and minimizing communication necessary between array chunks holding reactant concentrations and temperature fields. Hewlett Packard specific pragmas are inserted in order to help the

compiler with loop-dependent variable instances and loop-level parallelism across particular indices. As in Threat Analysis, automatic (compile time flags) parallelization by the compiler is thwarted by global variables within loops and function calls. Modifications to help the compiler recognize safe code segments to parallelize are mostly straightforward, but require understanding the application's higher level purpose and data dependencies.

Adjusting the number of grid points in the benchmark input parameter file provides an easy way to alter the various levels of parallelism. With more grid points, a higher fidelity solution for each mesh entity is calculated, affording the opportunity for more parallelism at the outer loop level. More memory traffic is also expected with finer grain mesh.

Looking for memory contention can be accomplished with the `pmon` process monitoring tool on the V-Class, and `cxperf` performance analysis tool on the X-Class. Both tools showed significant numbers of unexpected data cache misses coming from a frequently called subroutine invoking the `exp()` exponential function. A little searching in the `exp()` man page reveals a tip to try compiling with `+Olibcalls`, which uses millicode versions of math library routines. Subsequent runs showed much more reasonable data cache miss counts.

Additional investigation with `cxperf` on the X-Class shows a particular large iteration count loop incurring very high average memory latencies, indicating processors passing around the same cache lines. Previous experience on the part of Hewlett Packard technical consultants inspired closer inspection of a sum reduction within the loop. A little hand tuning cut the speed of the offending loop iteration in half. Such an optimization is completely transferable to a distributed memory implementation.

Table 3 shows execution times (seconds) for X- and V-Class runs. Performance gains can be attributed to cache residency and sufficient parallelism. For instance, a 301×301 grid size would take approximately 3 Megabytes, thus fitting into the data cache of four X-Class cpus and two V-Class cpus nicely.

Minimal pragma insertions and code reorganization were necessary for the base multithreaded implementation.

CPUs	**HP V2250**			**HP X2000**		
	Grid Size 201×201	**Grid Size 301×301**	**Grid Size 401×401**	**Grid Size 201×201**	**Grid Size 301×301**	**Grid Size 401×401**
1	12	82	247	44	115	356
2	8.41	49		25	66	198
4	5.6	33	98	14	43	123
8	4	23	66	9.3	31	79
16	3	16	74	6.25	30	57

Tab. 3: Execution Times (seconds) for Multithreaded Thermal Explosion

5 Conclusions

The Threat Analysis code requires algorithmic modification and some code reorganization for a baseline multithreaded version. Since Threat Analysis started off purely sequential, the underlying high-level algorithmic meaning had to be uncovered and implemented in a "compiler friendly" multithreaded manner. Once this is accomplished, we see nice speedups, given sufficiently large input threat sets to provide adequate opportunities for parallelism across outer loops. The multithreaded version of the sequential code is no harder to implement than a distributed memory version would be, and will continue to scale nicely as input data sets increase, given sufficient memory for oversized interval arrays. The X- and V-Class systems provide minimal, but useful, tools to help tune a multithreaded code within the GSM programming model.

The Thermal Explosion code also requires minor code reorganization and pragma insertion for a baseline multithreaded implementation to emerge from a working sequential version. As hoped, highly time-variable work imbalance per grid point results in reasonable performance, provided adequate parallelism is available.

Data required in our computations is kept sufficiently close to threads of execution on a given processor, such that special attention to data locality or load balancing does not appear not necessary for this benchmark.

With performance analysis tools, experience, and a bit of luck, time consuming portions of the both benchmark codes were identified and hand tuned for good overall performance. Choosing the right compiler flags, hand optimizations and code restructuring to enable compiler optimizations are equally valuable for an MPP implementation.

Acknowledgments

I wish to thank Caltech's Dr. Roy Williams for authoring the Thermal Explosion sequential code. Caltech's Dr. John Thornley provided major algorithmic and coding suggestions for the C3I benchmarks. Hewlett Packard's Kirby Collins was instrumental in characterizing initial V2250 scaling problems, and suggesting source code modifications to bring out the best in the HP compiler.

References

1. Metzger, R., VanVoorst, B., Pires, L. S., Jha, R., Au, W., Amin, M., Catanon, D. A., and Kumar, V., "The C3I Parallel Benchmark Suite—Introduction and Preliminary Results," in *Supercomputing '96*, Pittsburgh, PA, November 17–22, 1996.
2. "On the Nonlinear Stability and Detonability Limit of a Detonation Wave for a Model Three-step Chain-branching Reaction," *J. Fluid Mech.*, Vol. 339, pp. 89–112, 1997.