Parallelization Spectroscopy: Analysis of Thread-level Parallelism in HPC Programs

Arun Kejariwal
University of California, Irvine

Călin Caşcaval
IBM T.J. Watson Research Center

Abstract

In this paper, we present a thorough analysis of thread-level parallelism available in production High Performance Computing (HPC) codes. We survey a number of techniques that are commonly used for parallelization and classify all the loops in the applications studied using a sensitivity metric: how likely is a particular technique is successful in parallelizing the loop. We call this method parallelization spectroscopy. *Using parallelization spectroscopy, we show that in most of the benchmarks, at the loop level, more than >75% percent of the runtime is inherently parallel.*

1. Introduction

The need for high performance systems coupled with power constraints has driven the development of both homogeneous and heterogeneous multi-core chips. Examples include Intel's Nehalem and Sandy Bridge, IBM's Cell and POWER processors, and Sun's UltraSPARC T* family. Such systems require large scale (thread-level) parallel program execution, wherein the threads are mapped onto different physical processors. However, in practice, efficient exploitation of thread-level parallelism (TLP) is non-trivial. This, in part, is due to the lack of abstractions for expressing parallelism at the programming language level, lack of easy-to-use parallel programming models, limitations of compiler-driven program parallelization and many other practical limitations such as the threading overhead, destructive cache interference between the threads and non-graceful scaling of resources such as the memory bus bandwidth. Recently, there has been a large body of work addressing these issues as discussed below:

Software: There has been an increasing impetus in the development of new programming languages to efficiently capture the inherent parallelism early in the software development cycle. Examples include IBM's X10 [59], Sun's Fortress [15] and Cray's Chapel [10] languages. On the other hand, new programming models such as OpenMP [40] and PGAS [52] are being developed or extended to ease parallel programming. Further, parallel data structures [28] and libraries [6] are being developed to assist the component-based software development, a key to high achieving high productivity.

Hardware: As shown in [29], applications from recognition, mining, and synthesis (RMS) domains have small (parallel) tasks thereby limiting the speedup achievable via multithreading in software. This also requires architectural support for exploiting fine-grain TLP. Additionally, thread-level speculation (TLS) [49] and transactional memory (TM) [21] have been proposed as a means to extract optimistic concurrency from potentially parallel program regions.

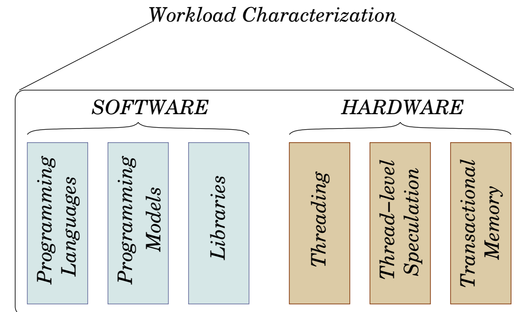

Figure 1. The role of workload characterization

Workload characterization (Figure 1) plays a critical role in guiding research and development of both software and hardware [7]. This stems from the fact that the introduction of any new idea into mainstream software or hardware is highly dependent on its applicability to existing and emerging workloads. In fact, based on workload characterization, there has been an increasing emphasis on the design of (i) new partitioned global address space (PGAS) languages such as UPC, X10 and Chapel, (ii) new domain-specific programming languages such as MATLAB for scientific computing, parallel version of SQL for database applications [22] and (iii) architectures [48, 3]. For this purpose, we present a detailed and practical analysis of the available TLP in production HPC codes. Specifically, we determine the coverage, defined as the percentage of the total execution time, of inherently parallel program regions such as parallel loops (or `DOALL` loops [34]). Additionally, we highlight the granularity, of the available parallelism, and we detail the factors inhibiting parallelization. We refer to this analysis as *parallelization spectroscopy*.

The main contributions of this paper are:

- A thorough characterization of the task level parallelism for loops in a large number of HPC workloads that are characteristic of production level codes;
- A synopsis of techniques used for parallellization; an integrated framework for workload characterization with respect to parallel execution, the parallelization spectroscopy, that was used for our characterization work;
- A realistic estimation of the speculation potential for scientific workloads; more than 75% of the execution time in these benchmarks is inherently parallel. To attain this parallelism we need enhanced compiler support or user annotations, but not necessarily speculation support. We also conclude that good parallel library development is critical for the performance of scientific codes;

A number of tools [55, 58] use techniques such as the parallelization spectroscopy to guide the selection of loops. In that context, some of the manual analyses presented in this paper have been automated, such as dependence profiling, reduction recognition, and profitability measures. Most of the other analysis techniques presented here can also be automated. However, the focus of this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00

paper is the detailed characterization of the HPC workloads, rather than the description of the tool.

The rest of the paper is organized as follows: Section 2 walks through the various facets of parallelization spectroscopy. Section 3 briefs the benchmarks used in this work. Experimental setup is described Section 4. Evaluation of the available thread-level parallelism in the benchmarks is presented in Section 5. Related work is discussed in Section 6. Finally, in Section 7 we conclude with directions for future work.

2. Parallelization Spectroscopy

In this section, we introduce the different dimensions of our spectroscopic analysis of loop-level parallelization of HPC codes.

First, we survey the set of techniques required for parallelizing loops in HPC codes, and we log the frequency of applicability of such techniques. We define a metric, denoted as $\mathcal{S}(\mathbf{L}, T)$, to measure the parallelization sensitivity of an application $\mathcal{P}$ with respect to a technique T.

$$\mathcal{S}(\mathbf{L}, T) = \frac{\text{Number of loops to which } T \text{ is applied}}{|\mathbf{L}|}$$

where $\mathbf{L}$ is the set of all the loops in $\mathcal{P}$. Note that $\mathcal{S}(\mathbf{L}, T) \in [0, 1]$. A high value of $\mathcal{S}(\mathbf{L}, T)$ signifies that T is required for the parallelization of a large number of loops in $\mathcal{P}$. However, $\mathcal{S}(\mathbf{L}, T)$ does not quantify the performance gain achievable based on the loops parallelized using T. Thus, we define the following metric:

$$\mathcal{S}_{cov}(\mathbf{L}, T) = \frac{\sum_{L_i \in \mathbf{L}_T} cov(L_i)}{\sum_{L_i \in \mathbf{L}} cov(L_i)}$$

where $\mathbf{L}_T$ is the set of loops ($\subset \mathbf{L}$) parallelized via application of T and $cov(L_i)$ denotes the coverage of loop L_i. A higher value of $\mathcal{S}_{cov}(\mathbf{L}, T)$ signifies that loop parallelization based on T has a large performance gain potential. Note that $\mathcal{S}_{cov}(\mathbf{L}, T) \in [0, 1]$. For many loops, as evidenced by the analysis presented in Section 5, more than one technique may be required for parallelization. In such cases, we define the following relational metric:

$$\mathcal{RelS}_{cov}(\mathbf{L}, T_j, T_k) = \frac{\sum_{L_i \in \mathbf{L}_{(T_j, T_k)}} cov(L_i)}{\sum_{L_i \in \mathbf{L}} cov(L_i)}$$

where $T_j \neq T_k \ \wedge \ T_j, T_k \in \mathbf{T}, \mathbf{T}$ is the set of all techniques and $\mathbf{L}_{(T_i, T_j)}$ is the set of loops ($\subset \mathbf{L}$) parallelized via application of both T_i and T_j. Note that $\mathcal{RelS}_{cov}(\mathbf{L}, T_j, T_k) \in [0, 1]$. The above metric can be easily extended to higher order (> 2) relations, by adjusting the subsets of loops for transformation sequences $T_0 \ldots T_n$: $\mathbf{L}_{(T_0 \ldots T_n)} = \{\mathbf{L} | T_0 \ldots T_n \text{were used to parallelize}\}$.

In practice, the achieved speeedup is subject to a multitude of factors such as (but not limited to) the underlying architecture and threading overhead. The $\mathcal{RelS}_{cov}$ metric provides a valuable guidance to programmers and compiler writers to select transformations sequences that provide maximum performance impact. Note that the order of the transformations may affect the performance as well. In this paper we do not consider the order in which the transformations were applied.

Second, we identify and characterize the bottlenecks such as I/O, which inhibit loop parallelization. These bottlenecks must be removed by user intervention, re-coding the application to use parallel I/O operations, as there are no automatic techniques to handle parallel I/O. And third, we assess amount of nested TLP in the HPC codes. For outer loops (in a given loop nest) with small number of iterations, it is critical to exploit, wherever possible, nested TLP. This is particularly important in light of the increasing number of hardware contexts in the emerging multi-core systems.

The following lists the techniques or transformations we considered for spectroscopy analysis:

- *Reduction:* It exploits commutative property of a computation such as accumulation, to drive loop parallelization. As shown in Section 5, reduction is widely applicable in parallelization of HPC codes;
- *Scalar/Array Privatization:* It is used to eliminate a loop-carried dependence by instantiating a local copy of the source of the loop-carried dependence in each iteration of the loop [32, 51]. Akin to reduction, privatization is also widely applicable in parallelization of HPC codes;
- *Loop Transformations:* A large variety of loop transformations such as (but not limited to) loop permutation have been proposed for (or to assist) loop parallelization[4]. We shall discuss their efficacy case by case for our workloads;
- *Symbolic Analysis:* Loop-carried dependences can be eliminated via symbolic analysis [19]. We assess the applicability of this technique;
- *Call-site Analysis:* In real codes, it is not uncommon that a hot loop may not be *intrinsically* amenable for parallelization. In such case, it is imperative to explore parallelizability at higher levels of abstraction. For this, we carried demand-driven multi-level call-site analysis of the function(s) containing the hot loop(s). Specifically, we traverse the call graph and identify whether the call site of such function(s) belongs to a parallel program region.

3. Benchmark Selection

Several limit studies have assessed the amount of inherent thread-level/task-level parallelism [41, 27, 26, 25]. These studies were primarily based on the SPEC CPU benchmarks [50]. Although the suite is widely used and is considered to be representative of a wide spectrum of application domains, the selection of the benchmarks has been a subject of discussion with respect to application balance et cetera [36, 42]. In light of this, we selected benchmarks from multiple industry-strength suites such as the Sequoia benchmark suite [45] from LLNL, publicly available codes such as CPMD [12] and production codes used in the DARPA HPCS program. The evaluation of available TLP presented in this paper complements prior work by sheding light on a wider set of applications.

Table 1 lists the benchmarks, their size in terms of number of lines of code, programming language and parallelization support in the original source code.

Benchmark	Lines of Code	Language	Parallelization Support
AMG	108169	C	MPI
CrystalMk	468	C	
IRSmk	457	C	
CPMD	194823	Fortran	OpenMP, MPI
POP	65654	Fortran90	OpenMP, MPI
UMT2K	19931	Fortran/C	OpenMP, MPI
RF-CTH	534382	Fortran/C	MPI
SPPM	20957	Fortran	OpenMP or MPI
HYCOM	32187	Fortran	OpenMP or MPI
Sweep3d	1952	Fortran	MPI

Table 1. Overview of the benchmarks

4. Experimental Setup

The analysis presented in Section 5 is empirical. In order to alleviate the artifacts of one system, we performed the experiments on two different systems, whose detailed configuration is given in Table 2. We compiled the applications listed in Table 1 using the IBM XLC v9/XLF v10 compiler (for the Power system) and gfortan/gcc 4.1.2 [16] (for the X86 sytem). We used the `-pg` option along with

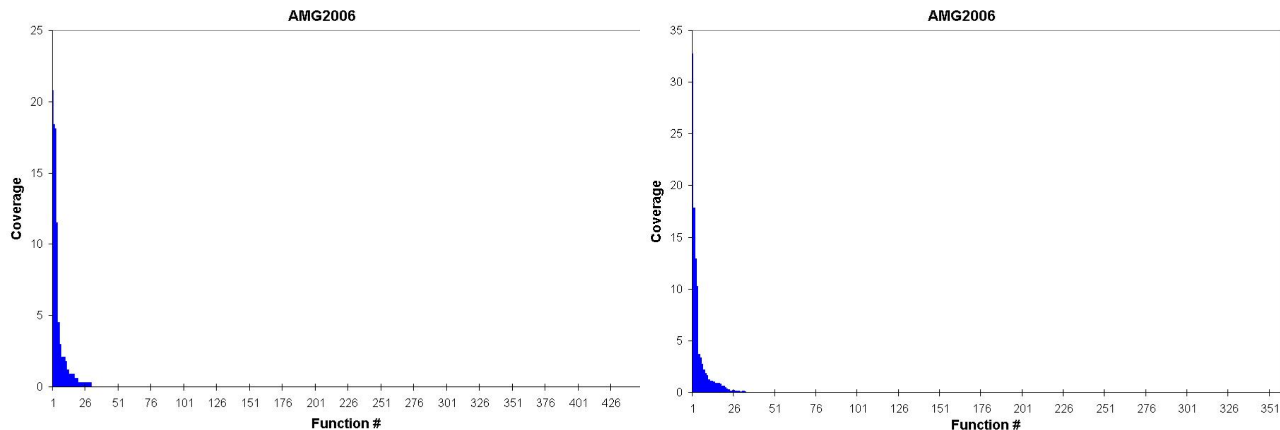

Figure 2. Function-level profile of AMG on (a) POWER5 (b) Xeon

other options and then used gprof [17] to obtain the function-level coverage profiles. For reproducibility of the results presented in this paper, the application specific compiler optimization flags and the run time commands used are reported in the respective subsections in Section 5. For applications parallelized using MPI and/or OpenMP directives, the results are presented for one MPI task or a single OpenMP thread, unless stated otherwise.

In order to be consistent with methodology employed in previous limit studies, based on the SPEC CPU benchmarks, we did not modify the source code of any application before compilation.

5. Parallelism Evaluation

In this section we present a detailed evaluation of available parallelism in production codes listed in Table 1. For this, we employed the following approach:

a) First, for each application, we breakdown the function-level coverage profile into three categories:[1] (1) inherently parallel (IP) program regions; (2) potentially parallel (PP) program regions and (3) "mostly" serial (MS) program regions. The coverage of IP serves as an upper bound on the speedup achievable via conventional multithreaded execution. The coverage of PP serves as an upper bound on the speedup achievable via multithreaded execution with support for explicit inter-thread synchronization and/or optimistic parallelization, such as TLS. In practice, the performance gain achievable is subject to a wide variety of factors such as cache interference between the different threads and threading overhead. A detailed analysis of these factors requires a precise machine model and is beyond the scope of this paper.
b) Second, we identify the bottlenecks which inhibit straightforward parallelization of program regions belonging to the PP category. We illustrate this with the help of code snippets.

The reminder of this section, presents the analysis of available parallelism on a case by case basis.

[1] The breakdown is similar to the classification of programs proposed by von Praun et al. based on dependence density [54].

Processor	Intel Xeon, 2.8 GHz	POWER5, 1.6 GHz
Memory	512 MB	3.8 GB
L1 Cache	8 KB	64 KB
L2 Cache	512 KB	32 MB
L3 Cache	None	32 MB
OS	Linux 2.6.9 Fedora Core 3	AIX 5.3

Table 2. Experimental Setup

5.1 AMG

AMG is an algebraic multigrid solver for linear systems arising from problems on unstructured grids [44]. The driver provided builds linear systems for various 3D problems. The code is written in ISO standard C. The purpose of the benchmark, as mentioned by LLNL, is to test single CPU performance and scaling efficiency. AMG is part of the Sequoia benchmark suite [45] from LLNL.

The compiler options used while compiling AMG were:

```
-O2 -DTIMER_USE_MPI -DHYPRE_NO_GLOBAL_PARTITION
```

Subsequently, we ran the binary with the following (default) options:

```
mpirun -np 1 amg2006 -r 6 6 6 -printstats
```

The function-level coverage profile of AMG on POWER5 and Xeon is shown in Figure 2. The total number of functions – the range of the x-axis in each profile – reported for each application correspond to the dynamically executed functions. This need *not* be equal to the number of functions in the source code. The latter can be, in part, ascribed to the following: (a) a function may not be executed for a given input data set and (b) functions may be inlined the compiler. The difference in the total number of functions in the two profiles is due to the difference in heuristics and phase ordering employed by the IBM XL and gcc compiler. For instance, the IBM XL and gfortran/gcc compilers employ different function inlining heuristics. This directly affects the total number of dynamically executed functions. Note that the two profiles shown in Figure 2 are very similar. This trend holds for all the applications we studied, which serves as an empirical evidence the results and analysis presented in this paper are not an artifact of the underlying architecture or a particular compiler. Due to space limitations, we are unable to include the coverage profile of all the applications.

We analyzed the loops in the top 5 hot functions to assess the inherent TLP in AMG. For instance, the loops in the hottest function `hypre_BoomerAMGRelax` (coverage of 20.8%) have the code as of the following loop (taken from file `hypre_BoomerAMGRelax.c`, line number 188):

```
for (i = 0; i < n; i++) {
   if (A_diag_data[A_diag_i[i]] != zero) {
      res = f_data[i];
      for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++) {
         ii = A_diag_j[jj];
         res -= A_diag_data[jj] * Vtemp_data[ii];
      }
      for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++) {
         ii = A_offd_j[jj];
         res -= A_offd_data[jj] * Vext_data[ii];
      }
      u_data[i] *= one_minus_weight;
      u_data[i] += relax_weight * res / A_diag_data[A_diag_i[i]];
   }
}
```

From the snippet we note that the loop is a `DOALL` loop, subject to privatization of variables such as `ii`, `jj` and `res`. The presence of a subscripted subscript (the read from the array `A_diag_data`) and/or a conditional does not necessarily inhibit the parallelization of the loop! Likewise, the hot loops in the function `hypre_BoomerAMGBuildCoarseOperator` (coverage of 18.4%), at line numbers 541, 748, 1121 and 1393 in the file `par_rap.c` are `DOALL`s. On further analysis we note that the hot loops in other functions such as `hypre_BoomerAMGBuildInterp` (coverage of 18.1%) and `hypre_CSRMatrixMatvec` (coverage of 11.5%) are also `DOALL`s. Overall, more than 75% of the total coverage belongs to the IP category.

Non-`DOALL` loops in AMG have a conditional dependence between the different iterations. For example, let us consider the following loop (taken from the file `par_interp.c`, line number 236):

```
for (i=0; i < num_cols_A_offd; i++) {
  for (j=A_ext_i[i]; j < A_ext_i[i+1]; j++) {
    k = A_ext_j[j];
    if (k >= col_1 && k < col_n) {
      A_ext_j[index] = k - col_1;
      A_ext_data[index++] = A_ext_data[j];
    }
    else {
      kc = hypre_BinarySearch(col_map_offd,k,num_cols_A_offd);
      if (kc > -1) {
        A_ext_j[index] = -kc-1;
        A_ext_data[index++] = A_ext_data[j];
      }
    }
  }
  A_ext_i[i] = index;
}
```

From the code snippet, we observe that the conditional increment of the variable `index` may induce a dependence between the (respective) writes to the arrays `A_ext_j` and `A_ext_data` in different iterations of the loop. Further, the variable `index` is neither an induction variable nor can it be privatized. Due to this, the loop is classified as a non-`DOALL` loop.[2] However, the upper bound of the aforementioned run time option is zero! Assuming that the default options are representative of the general case, we argue that such non-`DOALL` loops do not impact the parallel performance significantly.

5.2 Crystalmk

Crystalmk is a single CPU, C program intended to be an optimization and SIMD compiler challenge [46] and is part of the Sequoia benchmark suite [45] from LLNL. It consists of selected small portions of a large material strength package; however, the performance of this very set dominates the performance of the full package.

Based on our analysis of the function-level coverage profile of Crystalmk we note that the function `Crystal_Cholesky` accounts for the largest coverage – 37.17% on the POWER5 – of all the functions. Let us consider the main loop of the function `Crystal_Cholesky`, taken from file `Crystal_Cholesky.c`, line number 33.

[2] The call to the function `hypre_BinarySearch` does not have side effects.

```
L1: for ( i = 1; i < nSlip; i++){
      fdot = 0.0;
L2:   for ( k = 0; k < i; k++)
        fdot += a[i][k] * a[k][i];
      a[i][i] = a[i][i] - fdot;
L3:   for ( j = i+1; j < nSlip; j++){
        fdot = 0.0;
L4:     for ( k = 0; k < i; k++)
          fdot += a[i][k] * a[k][j];
        a[i][j] = a[i][j] - fdot;
        fdot = 0.0;
L5:     for ( k = 0; k < i; k++)
          fdot += a[j][k] * a[k][i];
        a[j][i] = ( a[j][i] - fdot) / a[i][i];
      }
    }
```

From the code snippet we note that the outer loop L1 is a non-`DOALL` loop. For example, the array element `a[2][1]` is written to in iteration 1 and is then read in iteration 2 of the loop `L1`. However, the inner loops L2 and L3 are `DOALL`s. Loop L2 can be parallelized using OpenMP-type reduction of the variable `fdot`, whereas loop L3 can be parallelized via privatization of the variable `fdot`. The key to parallelization of loop L3 is the exploitation of the condition $k < i$ in the header of the loops L4 and L5. This guarantees that there is no dependence between the iterations of the loop L3 (see Figure 3 for the memory access pattern).

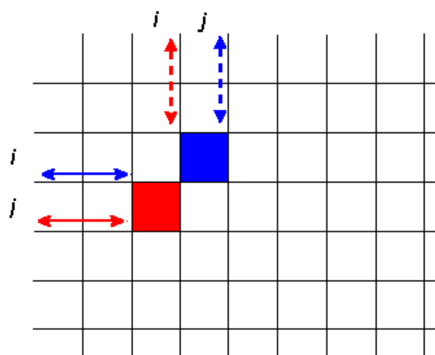

Figure 3. Memory access pattern for iteration i of loop L3

In Figure 3, the matrix represents the array `a`. The shaded blocks correspond to the elements of `a` written in the first iteration of the loop L3. Let us consider the blue colored block which corresponds to `a[i][j]`. The blue arrows represent the set of elements of the array `a` read for computing `fdot` which is then subtracted from `a[i][j]`. As `j` increases, the dashed arrow – which corresponds to `a[k][j]` in loop L4 – shifts to the right, whereas the solid arrow – which corresponds to `a[i][k]` in Loop L4 – remains "fixed". Similarly, in the case of the red colored block which corresponds to `a[j][i]`, the dashed arrow moves downwards and the solid arrow remains "fixed". On analysis, we note the read and writes in loops L4 and L5 do *not* introduce a loop-carried dependence between the iterations of loop L3.

Based on run-time analysis of the loop above, we find that the value of the variable `nSlip` is 12. Given this, the number of iterations of loop L3, on an average, is 5.5. Hence, we argue that if the number of processors is less than 6, then it is more profitable to exploit TLP at the level of loop L3. Our recommendation is based on the fact that the L3 is a `DOALL` loop. If the number of processors is more than 5 then TLP and speculative thread-level parallelism may be exploited at the levels of loops L3 and L1 respectively.

On the other hand, the hot loop in the function `Crystal_div` (which has a coverage of 21.1% on POWER5) is a `DOALL` loop (the loop is shown below). Interestingly, the library function `_pow` accounts for a coverage of 13.3% on the POWER5! This suggests that lack of parallelization of such library routines would limit the performance gain achievable via parallelization of only the source code.

```
for (n = 0; n < nSlip; n++) {
  tauN[n] = tau[n];
  for (m = 0; m < nSlip; m++) {
    bor_s_tmp = dtdg[n][m]* deltaTime;
    tauN[n] +=  bor_s_tmp * dSlipRate[m] ;
    matrix[n][m] = (-bor_s_tmp + dtcdgd[n][m])*bor_array[n];
  }
  err[n] = tauN[n] - tauc[n];
  rhs[n] = err[n] * bor_array[n];
}
```

5.3 IRSmk

IRSmk [47] is part of the Sequoia benchmark suite [45] from LLNL. The purpose of the benchmark, as stated by LLNL, is to assess the scaling efficiency and single processor performance assessment. We compiled the benchmark using the `gcc-4.1.2` compiler and with following options: `-c -O3 -pg`.

The benchmark code comes along with three input data sets viz., `irsmk_input_25, irsmk_input_25` and `irsmk_input_100`. We used all the three input data sets for our experiments. The binary was executed using the `./IRSmk` command. From the figure we observe that the first function (`rmatmult3`) accounts for over 99% of the total execution time. The only loop in the function mentioned above, taken from file `rmatmult3.c`, line number 79 (shown below), has a coverage of 99% on both the systems.

```
L1: for ( kk = kmin ; kk < kmax ; kk++ ) {
L2:   for ( jj = jmin ; jj < jmax ; jj++ ) {
L3:     for ( ii = imin ; ii < imax ; ii++ ) {
          i = ii + jj * jp + kk * kp ;
          b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] +
                 dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i] * xdcr[i] +
                 dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i] +
                 cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i] +
                 ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i] +
                 cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i] +
                 ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i] +
                 ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i] +
                 ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i] ;
      }
    }
  }
```

On analyzing the code snippet we note that the outer loop is a `DOALL` loop, subject to privatization of variables such as `kk, jj, ii` and `i`. Thus, based on code analysis, we classify IRSmk under the IP category.

Based on run-time analysis, we find that the iteration count of each loop in the triply nested loop is 100. Table 3 reports our recommended loop nesting level for multithreading based on the number of processors. Arguably, loop L3 can also be multithreaded if the number of processors is $> 10^4$. However, this may not be profitable due to small amount of computation in the loop body.

# of processors	Loop-level
≤ 100	L1 only
$100 < \& \leq 10^4$	L1 and L2

Table 3. Granularity of multithreading for IRSmk

5.4 CPMD

The CPMD code is a plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics (MD) [12]. We compiled the source code using the gfortran-4.1.2 compiler with the following options:

```
-O2 -fcray-pointer -fsecond-underscore -pg
```

and used the following libraries

```
-llamf77mpi -lmpi -llam -lpthread -lblas -llapack
```

The CPMD code internally invokes routines from the LAPACK package [31]. We used version 3.1.1 of the same. We ran the binary on a Xeon-based 4 processor system. We used both the input data sets provided with the distribution and used the LAM/MPI [30] for running the binary on the 4-processor system.

On analyzing the function-level coverage profiles we note that the coverage of the hottest function is 31.28% and 41.91% for input data sets 1 and 2 respectively. The loops in the hottest subroutine `fftstp` are of the type shown on the right hand side `gfft.f`, line number 70).

On analyzing the code snippet we note that loops L3 and L5 are `DOALL`s, subject to privatization of variables such as `r` and `s`. Likewise, loops L2 and L4 are also `DOALL`s, subject to IVE (induction variable elimination) [37] and `atn` $> 3\times$ `after`. The latter stems from the following:

1.
$$\texttt{nout4} = \texttt{nout1} + 3 \times \texttt{after} \quad (1)$$
$$\texttt{nout3} = \texttt{nout1} + 2 \times \texttt{after} \quad (2)$$
$$\texttt{nout2} = \texttt{nout1} + \texttt{after} \quad (3)$$
2. The variable `nout1` increases monotonically with increasing value of `ib`.

From above and the fact that `nout` is incremented by `atn` in each iteration, we conclude that the writes to the array `zout` do not alias if `atn` $> 3\times$ `after`. Next, let us analyze the outermost loop L1. We observe that along the `then` as well as the `else` branch of the conditional:

- The initialization of `nout1`: `nout1=ia-atn`, is the same.
- The variables `nout1, nout2, nout3` and `nout3` are computed in the same fashion.
- The lower and upper bounds of loops L2 and L3 are the same as that of the loops L4 and L5.

```
L1: do 4000,ia=2,after
      ias=ia-1
      if (2*ias.eq.after) then
        nin1=ia-after
        nout1=ia-atn
L2:     do 4010,ib=1,before
          nin1=nin1+after
          nin2=nin1+atb
          nin3=nin2+atb
          nin4=nin3+atb
          nout1=nout1+atn
          nout2=nout1+after
          nout3=nout2+after
          nout4=nout3+after
L3:       do 4010,j=1,nfft
            ...
            zout(1,j,nout1) = r + s
            zout(1,j,nout3) = r - s
            ...
            zout(1,j,nout2) = r - s
            zout(1,j,nout4) = r + s
            ...
            zout(2,j,nout1) = r + s
            zout(2,j,nout3) = r - s
            ...
            zout(2,j,nout2) = r + s
            zout(2,j,nout4) = r - s
    4010 continue
      else
        ...
        nin1=ia-after
        nout1=ia-atn
L4:     do 4020,ib=1,before
          nin1=nin1+after
          nin2=nin1+atb
          nin3=nin2+atb
          nin4=nin3+atb
          nout1=nout1+atn
          nout2=nout1+after
          nout3=nout2+after
          nout4=nout3+after
L5:       do 4020,j=1,nfft
            ...
            zout(1,j,nout1) = r + s
            zout(1,j,nout3) = r - s
            ...
            zout(1,j,nout2) = r - s
            zout(1,j,nout4) = r + s
            ...
            zout(2,j,nout1) = r + s
            zout(2,j,nout3) = r - s
            ...
            zout(2,j,nout2) = r + s
            zout(2,j,nout4) = r - s
    4020 continue
      endif
    4000  continue
```

In light of the above, the condition for no aliasing of writes to the array `zout` from L1 perspective is the same as that for loops L2 and L4: `atn` $> 3\times$ `after`. This condition can be used as a basis for loop versioning whereby a parallelized version of the outermost loop can be invoked at run-time subject to the satisfiability of the above condition.

The inability of a compiler to determine the satisfiability of the condition mentioned above may suggest to include the coverage of the outermost loop, minus the coverage of inner parallel loops, under the PP category. However, on further program analysis, we observe that the function `fftstp` is called in the parallel loop at line 53 in the file `mltfft.f`. The loop is parallelized using OpenMP pragmas in the *original* source code. Therefore, the coverage of the function `fftstp` should be counted as part of the coverage of parallel regions of CPMD. Similarly, the functions `fftpre` and `fftrot`, which have coverages of 13.81% and 7.83%, are called from the above loop. Overall, our analysis shows that more than 75% of the total coverage is inherently parallel. Further, the upper bound of the loop at line 53 in the file `mltfft.f` is equal to the number of processors (`NCPUS`). Thus, in the current case, nested multithreading is unwarranted!

Lastly, the function `zazzero` (coverage of 12.45%) consists of a loop wherein the elements of an array are set to zero. Thus, parallelization of the library routine `memset` can help in obtaining better performance.

5.5 POP

POP is an ocean modeling code, written in Fortran90, developed at Los Alamos National Lab. Prior to building the binary, we installed LAM/MPI [30], version 7.1.4 and the `netcdf` library [39], version 3.6.2. Then, we compiled POP using the IBM `xlf90` compiler with `-O3 -pg -qsave -qmaxmem=131072 -q64 -qsuffix=f=f90 -qfree=f90` options and used `mpif77` for linking. We ran POP on a single processor using the `./pop` command and using a real dataset.

On analyzing the function-level coverage profile we see that the maximum coverage of an individual function – the function `state` in the module `state_mod` – is 30.28%. Unlike the hot functions in the benchmarks analyzed so far, `state` does not contain any loops! `state` consists of a 3-way and a 4-way `select` statements. Conceivably, the function can be parallelized via multipath execution [2] in conjunction with hardware/software support for termination of a wrongly executed path. Akin to the methodology followed for analysis of CPMD, we traced the calling context of `state` to explore TLP at higher level of abstraction.

filename:line number	Caller function	Called inside a `DOALL` loop?
advection.f90:1406,1413,1463,1470	advt	✗
advection.f90:1479	baroclinic_driver	✗
baroclinic.f90:574	baroclinic_correct_adjust	✔
baroclinic.f90:926	hdifft_gm	✗
hmix_gm.f90:716,801,1663	hdifft_gm	✗
initial.f90:706,710	init_ts	✔
step_mod.f90:495,499	step	✔
vertical_mix.f90:1472,1474,1519	convad	✗
vmix_const.f90:216,218	vmix_coeffs_const	✗
vmix_kpp.f90:1019,1825	bldepth	✗
vmix_kpp.f90:1835,1977	ddmix	✗
vmix_kpp.f90:1979,1981	buoydiff	✗
vmix_rich.f90:300	vmix_coeffs_rich	✗

Table 4. Calling context of the function `state`

Table 4 details the calling context of `state` and reports whether it is called inside a `DOALL` loop, parallelized in the *original* source code using OpenMP pragmas. For example, the call to `state` in `initial.f90` is shown below:

```
!$OMP PARALLEL DO PRIVATE(iblock, k, this_block)
do iblock = 1,nblocks_clinic
  this_block = get_block(blocks_clinic(iblock),iblock)
  do k=1,km
    call state(k,k,TRACER(:,:,k,1,curtime,iblock), &
                   TRACER(:,:,k,2,curtime,iblock), &
                   this_block, &
                   RHOOUT=RHO(:,:,k,curtime,iblock))
    call state(k,k,TRACER(:,:,k,1,oldtime,iblock), &
                   TRACER(:,:,k,2,oldtime,iblock), &
                   this_block, &
    RHOOUT=RHO(:,:,k,oldtime,iblock))
enddo
enddo ! block loop
!$OMP END PARALLEL DO
```

From Table 4 we note that 5 (out of 26) contexts correspond to the IP category. Next, we traced second, third and so on levels of the calling context. For example, `state` is called by the function `advt` in `advection.f90`, `advt` is called by the function `tracer_update`. The latter is called inside a `DOALL` loop in the function `baroclinic_driver`. Thus, the first five calling contexts correspond to the IP category. Likewise, `state` is called by the function `vmix_coeffs_const` in `vmix_const.f90`, `vmix_coeffs_const` is called by the function `vmix_coeffs` which is in turn called inside a `DOALL` loop in the function `baroclinic_driver`. Based on the above analysis, we find that 22 contexts correspond to the IP category.

The second hottest function `hdiffu_aniso` (coverage of 13.09%) in the module `hmix_aniso` consists of mostly `DOALL` loops – at line numbers 691, 718 and 922 in the file `hmix_aniso.f90`. Similarly, most of the loops in the function `hdifft_gm` (coverage of 12.19%) in the module `hmix_gm` are `DOALL`s, e.g., the outermost loop at line number 1232 in the file `hmix_gm.f90` (the code of the loop – 194 lines – could not be included owing to space limitations) is a `DOALL` loop. Based on analyzing the top 10 hot functions, we conclude that more than 75% of the total coverage of POP is inherently parallel.

```
WORK1 = p5*(DZT(:,:,k,bid)+DZT(:,:,k+1,bid))* &
        KMASK*TAREA_R(:,:,bid)* &
        (DZT(:,:,k ,bid)*p25*KAPPA(:,:,kbt ,bid)* &
        (DZTE*HYX (:,:,bid)*SLX(:,:,ieast, kbt ,bid)**2 &
       + DZTW*HYXW(:,:,bid)*SLX(:,:,iwest, kbt ,bid)**2 &
       + DZTN*HXY (:,:,bid)*SLY(:,:,jnorth,kbt ,bid)**2 &
       + DZTS*HXYS(:,:,bid)*SLY(:,:,jsouth,kbt ,bid)**2) &
       + DZT(:,:,k+1,bid)*p25*KAPPA(:,:,kbt2,bid)* &
        (DZTEP*HYX (:,:,bid)*SLX(:,:,ieast, kbt2,bid)**2 &
       + DZTWP*HYXW(:,:,bid)*SLX(:,:,iwest, kbt2,bid)**2 &
       + DZTNP*HXY (:,:,bid)*SLY(:,:,jnorth,kbt2,bid)**2 &
       + DZTSP*HXYS(:,:,bid)*SLY(:,:,jsouth,kbt2,bid)**2))
```

Lastly, the code makes extensive use of Fortran90 intrinsics as shown above (taken from the file `hmix_gm.f90` line number 1137). The above computation involves matrix-scalar and matrix-matrix multiplication. Each intrinsic is "unfolded" into nested loop by the front end. The resulting loops can be executed in parallel as there does not exist any dependence between them. The coverage corresponding to such intrinsics corresponds to the PP category.

5.6 UMT2K

The UMT benchmark is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. UMT 1.2, referred to as UMT2K for clarity, includes features that are commonly found in large LLNL applications.

We compiled UMT2K using the IBM `xlf90` and `xlc` compilers with the `-O3 -q64 -qnosave -pg` options. For linking, we used `mpif77` and `mpicc` (of the LAM/MPI distribution). Then, we ran the binary on a POWER5-based system using the following command: `./umt2k -procs 1`.

On analyzing the function-level coverage profile we note that the function (`snswp3d`) accounts for >90% of the total coverage. The function consists of 9 outermost loops (at line numbers 221, 226, 235, 275, 299, 306, 316, 356 and 553). On analysis we note that the loops at lines 235, 316 and 356 are not parallel and the rest are `DOALL`s. However, the inner loops in the three non-`DOALL` loops are `DOALL` loops.

For instance, the loop at line 373 in the file snswp3d.c is a DOALL loop (see below) and is inside the loop at line number 356. Likewise, the other inner loops inside this outermost loop are DOALL loops. Based on this, we ascribe the coverage of the outermost loop, minus the coverage of the inner DOALL loops, to the PP category.

```
for (ip = 1; ip <= npart; ++ip) {
  /* Compute Sources */
  sigvx = sigvol_ref(ip, ix); // DELINQUENT LOAD
  sigvx2 = sigvx * sigvx;
  sface = sosf_ref(ip, iface);
  szone = sosz_ref(ip, iz);
  sosc1 = qc_ref(ip, ic);
  sosc2 = qc_ref(ip, ic2);

  sedgex = half * (sosc1 - sosc2);
  sosfpzx_ref(ip) = third * sigvx2 *
      (sface + szone - two * sosc1) / (sigvx2 + afpzm*afpzm);
  sospezx_ref(ip) = third * sigvx2 *
      (szone - sedgex - sosc1) /     (sigvx2 + apezm * apezm);
  source = two * sigvx * sosc1 - apezsum * sospezx_ref(ip) -
      afpzsum * sosfpzx_ref(ip);

  /*  Flux calculation */
  tauwt = (afezm * afezm) * (three * afezm + four * sigvx ) /
      (two*sigvx * ((afezm*afezm) + afezm*sigvx + two*sigvx*sigvx));
  denom = one / (asum * ((asum + two*afezm*tauwt) + two*sigvx));
  ybase = denom * (asum + two*afezm*tauwt);
  xbase = denom * ( (asum + two*afezm*tauwt) - two*sigvx*tauwt);
  sdifflim = sedgex + third * (sface - sedgex - sosc1) *
      afepm*sigvx / (afezm*afezm + sigvx2);
  stet = denom * asum * (source + two*afezm*sdifflim);
  sfez = stet + denom*asum*(tauwt*source - (asum + two*sigvx)*sdifflim);

  /* Calculate average angular flux in each tet (PSIT) */
  psit_ref(ip) = stet + ybase * psi_inc_ref(ip, ix);
  psifez = sfez + xbase * psi_inc_ref(ip, ix);
  tpsic_ref(ip, ic) = tpsic_ref(ip, ic) + tetwtx * psit_ref(ip);
  psi_inc_ref(ip, iexit) = psi_inc_ref(ip, iexit) +  two*afezm*psifez;
}
```

As mentioned earlier, the profitability of speculative parallelization of non-DOALL outermost loops is subject to, say (but not limited to), the dependence properties such as the minimum dependence distance [5]. For the outermost loop at line number 356, the minimum dependence distance is very small. Consequently, exploitation of TLP at the inner loop level may be more profitable in the current context.

5.7 RF-CTH

CTH is a code used to explore the effects of strong shock waves on a variety of materials using many different models.

We compiled RF-CTH using the IBM xlf and xlc compilers and executed it on a POWER5 machine with small1 and small2 input data sets. The datasets were provided along with the source code distribution. The run is done in two steps: first, the input is processed using rf-cthgen and then the processed input is fed to the binary rfcth. Next, we present the evaluation of available TLP in the latter.

We analyzed the top 25 hot functions which account for 90% of the total coverage. Akin to other benchmarks, the inner loops in these functions are DOALL loops. For example, let us consider the loop shown below, taken from the file erpy.F, line number 1551.

```
DO 6020  JJ=1,JMAX
  IF (NDXM.LT.0)  THEN
    VX(1,JJ)=PZERO
    VY(1,JJ)=PZERO
    IF (IGM.GE.30)  VZ(1,JJ)=PZERO
    IF (IHBXB(IAMBLK).EQ.0)  THEN
      IF (IDIOXB(IAMBLK).EQ.0)  THEN
        VX(2,JJ)=PZERO
        VX(1,JJ)=-VX(3,JJ)
      ELSEIF (IDIOXB(IAMBLK).EQ.1)  THEN
        IF (VX(2,JJ).LE.PZERO)  THEN
          VX(2,JJ)=PZERO
          VX(1,JJ)=-VX(3,JJ)
        ELSE
          VX(1,JJ)=VX(2,JJ)
        ENDIF
      ELSEIF (IDIOXB(IAMBLK).EQ.3) THEN
        IFLG=0
        IF (VX(2,JJ).LT.PZERO)  THEN
          DO 6030  NN=1,NXWB
            IF (Y(JJ).GE.YSWIN(1,NN) .AND.
     &          Y(JJ).LT.YSWIN(2,NN) .AND.
     &          Z(KPLANE).GE.ZSWIN(1,NN) .AND.
     &          Z(KPLANE).LT.ZSWIN(2,NN))  THEN
              IFLG=1
            ENDIF
 6030     CONTINUE
        ENDIF
        IF (IFLG.EQ.1)  THEN
         VX(1,JJ)=VX(2,JJ)
        ELSE
          VX(2,JJ)=PZERO
          VX(1,JJ)=-VX(3,JJ)
        ENDIF
      ENDIF
    ELSE
      VX(1,JJ)=VX(2,JJ)
    ENDIF
  ENDIF
  IF (NDXP.LT.0)  THEN
    VY(IMAX,JJ)=PZERO
    IF (IGM.GE.30)  VZ(IMAX,JJ)=PZERO
    IF (IHBXT(IAMBLK).EQ.0)  THEN
      IF (IDIOXT(IAMBLK).EQ.0)  THEN
        VX(IMAX,JJ)=PZERO
      ELSEIF (IDIOXT(IAMBLK).EQ.1)  THEN
        VX(IMAX,JJ)=MIN(VX(IMAX,JJ),PZERO)
      ELSEIF (IDIOXT(IAMBLK).EQ.3) THEN
        IFLG=0
        IF (VX(IMAX,JJ).GT.PZERO)  THEN
          DO 6040  NN=1,NXWT
            IF (Y(JJ).GE.YSWIN(1,NN+10) .AND.
     &          Y(JJ).LT.YSWIN(2,NN+10) .AND.
     &          Z(KPLANE).GE.ZSWIN(1,NN+10) .AND.
     &          Z(KPLANE).LT.ZSWIN(2,NN+10))  THEN
              IFLG=1
            ENDIF
 6040     CONTINUE
        ENDIF
        IF (IFLG.EQ.0)  THEN
          VX(IMAX,JJ)=PZERO
        ENDIF
      ENDIF
    ENDIF
  ENDIF
6020 CONTINUE
```

On analyzing the code snippet, we observe that there does not exist aliasing between the writes to the array VX in the different iterations. Also, the variable IFLG is local to each iteration. Thus, the loop is a DOALL loop. Likewise, the multi-way loop [43] at line 132 in the file erpy.F is also a DOALL loop. In contrast, the multi-way loop at line 263 in the same file cannot be auto-parallelized due to I/O – call to WRITE at line 376 – in the loop body. Thus, the coverage of this loop, minus the coverage of the inner parallel loops, is classified under the PP category.

All the loops in the functions convcy (coverage of 11.8%) and elsg (coverage of 7%) are of the type as the loop in the code snippet shown above and are DOALL loops. Overall, based on the loops we analyzed, more 65% of the total coverage of rfcth is inherently parallel.

The coverage reported above is an upper bound on the speedup achievable via vanilla multithreading. However, in practice, we find that it is not profitable to multithread many DOALL loops. This is due to their low coverage per invocation. In such cases, the performance gain achieved via multithreaded execution is offset by the threading overhead. For example, the function erfays has a coverage of 1.5% and is called 3755500 times. Given this and the configuration listed in Table 2, the run time of the function is approximately 8.7K cycles on an average. For simplicity, assuming uniform distribution of the run time between the different iterations of the loop, the run

time each loop spans for < 800 cycles which makes multithreaded execution of such loops non-profitable. This highlights the need for exploiting TLP at higher levels akin to CPMD and POP.

5.8 SPPM

SPPM is a simplified version of PPM, the Piecewise-Parabolic method. It contains a nonlinear Riemann solver and a careful computation of the Courant time step limit. We compiled SPPM using the IBM `xlf` compiler with the `-O3 -qhot -qarch=pwr5 -qnosave -qfixed=132 -qmaxmem=-1 -qautodbl=dbl4 -pg` options and ran the binary on POWER5.

```
do 3000 i = -nbdy+3,n+nbdy-1
   rpll = rplusr(i-1) - xf(i-1)*(drplus(i-1) - xf1(i-1)*rplus6(i-1))
   rmll = rmnusr(i-1) - xf(i-1)*(drmnus(i-1) - xf1(i-1)*rmnus6(i-1))
   uxll = ux(i-1) + 0.5*(rpll + rmll)
   pll = p(i-1) + 0.5*c(i-1)*(rpll - rmll)
   diffux = ux(i) - ux(i-1)
   sux = sign (1.0e+00, diffux)
   diffl = sux * (uxll - ux(i-1))
   diffr = sux * (ux(i) - uxll)

   if (diffl .lt. 0.0e+00) uxll = ux(i-1)
   if (diffr .lt. 0.0e+00) uxll = ux(i)
   ...
1: pavl(i) = max (smallp, (prl + dpavl * c(i)))
2: uxavl(i) = uxrl + dpavl

   wllfac = gamma1 * pll + gammp1*pavl(i)
   hrholl = 0.5 * rho(i-1)
   wll = sqrt (hrholl * wllfac)
   dpdull = wll * wllfac / (wllfac - hgamp1*(pavl(i) - pll))
   wrlfac = gamma1*prl + gammp1 * pavl(i)
   hrhorl = 0.5 * rho(i)
   wrl = sqrt (hrhorl * wrlfac)
   dpdurl = wrl * wrlfac / (wrlfac - hgamp1*(pavl(i) - prl))
   ustrll = uxll - (pavl(i) - pll) / wll
   ustrrl = uxrl + (pavl(i) - prl) / wrl
   thyng = (ustrrl - ustrll) * dpdurl / (dpdurl+dpdull)

3: uxavl(i) = ustrll + thyng
4: pavl(i) = max (smallp, (pavl(i) - thyng * dpdull))

   wllfac = gamma1*pll + gammp1*pavl(i)
   hrholl = 0.5 * rho(i-1)
   wll = sqrt (hrholl * wllfac)
   dpdull = wll * wllfac / (wllfac - hgamp1*(pavl(i) - pll))
   wrlfac = gamma1 * prl + gammp1 * pavl(i)
   hrhorl = 0.5 * rho(i)
   wrl = sqrt (hrhorl * wrlfac)
   dpdurl = wrl * wrlfac / (wrlfac - hgamp1*(pavl(i) - prl))
   ustrll = uxll - (pavl(i) - pll) / wll
   ustrrl = uxrl + (pavl(i) - prl) / wrl
   thyng = (ustrrl - ustrll) * dpdurl / (dpdurl + dpdull)

5: uxavl(i) = ustrll + thyng
6: pavl(i) = max (smallp, (pavl(i) - thyng*dpdull))

   uxpavl(i) = uxavl(i) * pavl(i)
   dvoll(i) = dt * uxavl(i)
   xnul(i) = xl(i) + dvoll(i)
3000 continue
```

We analyzed the top 6 hot functions of SPPM which account for a coverage of 94.19%. The hottest function `sppm` has a coverage of 30.61%. There are 12 outermost loops in this function and all of them are DOALLs. For illustration, the largest (in terms of lines of code) loop in the function sppm is shown above (taken from sppm.f, line number 703). On analyzing the code snippet, we note that the loop is infact a `DOALL` loop, subject to privatization of the scalar variables such as `rpll, rmll and pll`. There is no loop-carried dependence based on the write to the arrays such as `uxavl, pavl and uxpavl`. Beyond parallelization, higher speedup can be achieved by optimizing each iteration of the loop. For instance, the writes to the array `uxavl`, statements 1 and 3, can be eliminated as the corresponding values are overwritten by the write to `uxavl` in statement 5. The writes to the array `pavl` in statements 2 and 4 can be memory-renamed [37], e.g., statement 2 can be rewritten as:

```
TEMP = max (smallp, (pavl(i) - thyng * dpdull))
```

and then replace the reads to `pavl(i)` between statements 2 and 4 by the scalar `TEMP`. The write to `pavl` in statement 6 cannot be eliminated as the array `pavl` is not local to the loop.

The function `difuze`, at line 1135 in the file `sppm.f` and with a coverage of 14.87%, consists of a singly nested `DOALL` loop. Likewise, the function `interf`, at line 1725 in the file `sppm.f` and with a coveage of 13.37%, consists of a single nested `DOALL` loop. Overall, we note that more than 55% of the total coverage belongs to the IP category. On further analysis, we find that the intrinsics[3] `__vrec_GR` and `__vsqrt_GR` account for a coverage of 16.72% and 13.37% respectively. This suggests that further parallelization of SPPM is subject to the paralelization of the library calls mentioned above.

5.9 HYCOM

HYCOM is a Hybrid Coordinate Ocean ModeL developed from MICOM (Miami Isopycnic Coordinate Ocean Model) and NLOM (Navy Layered Ocean Model) by a Consortium of LANL, NRL and University of Miami [23].

We compiled the source code using the IBM `xlf` compiler with the following options: `-O3 -pg -qmaxdata:0x80000000/dsa -qstrict -qtune=pwr5 -qcache=auto -qspillsize=32000 -q64 -qfixed -qrealsize=8 -qintsize=4`. Subsequently, We ran HYCOM, using the following command `./hycom.single`, on a POWER5 processor.

We analyzed the top 15 functions which account for a coverage of 80%. Let us first consider the hottest function `momtum`. On analysis we find that all the outermost loops in the function are `DOALL` loops. As a matter of fact, majority of them are parallelized using the OpenMP pragmas in the *original* source code. Examples include the `DOALL` loop in `momtum.f:459` where most of the array accesses are not aliased and would benefit from scalar privatization.

Outermost loops in the second hottest function `.mxkppaij` (coverage of 9.3%) have dependence distance of 1 and are therefore classified under the MS category (recall that the systems listed in Table 2 do not have support for data value speculation (DVS)). In other functions, the outermost loops are parallelized in the *original* source code using OpenMP pragmas or are inherently parallel otherwise.

Lastly, on analyzing the function-level coverage profile we note that the library calls `.__mod_advem_NMOD_advem, ._atan2` and `._exp` account for 5.7%, 5.4% and 4.3% of the total execution time respectively. This suggests that parallelization of the above library calls bear a large potential for speeding up HYCOM.

5.10 Sweep3d

SWEEP3D represents the heart of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem.

We compiled SPPM using the IBM `xlf` compiler with the `-O3 -qhot -qarch=pwr5 -pg` options and ran the binary on POWER5. On analyzing the function-level coverage profile, we note that the hottest function (`sweep`) accounts for more than 90% of the total coverage. The function has a total of 67 loops, of which 53 are inner `DOALLs` loops. The loop at line 353 is parallelized using pragmas in the *original* source code. The loop at line 326, which contains the loop above, is also a `DOALL` loop, subject to IVE and scalar privatization. Loops at line numbers 217, 168 and 131 (outermost) contain the loop above could not be parallelized due to the presence of function calls.

Although the loop at line number 353 is already parallelized using OpenMP pragmas, it should not be "disregarded" for analysis while evaluating the available parallelism. This stems from the need for exploiting nested TLP which in turn is driven by the increasing number of cores on a chip [24]. For example, let us consider the loop at line number 416 shown below). The loop is inside the already parallelized loop discussed above.

From the code snippet we note that the writes to the arrays `phi, phijb` and `phikb` do not induce a loop-carried dependence. The loop is a `DOALL` loop subject to scalar privatization and application

[3] The instrisics are mapped on to optimized library calls.

of IVE on the `jfixed`. Overall, $> 75\%$ of the total coverage is inherently parallel.[4]

```
! DO PARALLEL
...
DO i = i0, i1, i2
  ci =  mu(m)*hi(i)

  dl = ( sigt(i,j,k) + ci + cj + ck )
  ti = 1.0 / dl
  ql = ( phi(i) + ci*phiir + cj*phijb(i,lk,mi) + ck*phikb(i,j,mi) )
  phi(i) = ql * ti

  ti = 2.0d+0*phi(i) - phiir
  tj = 2.0d+0*phi(i) - phijb(i,lk,mi)
  tk = 2.0d+0*phi(i) - phikb(i,j,mi)

  ifixed = 0
111       continue
  if (ti .lt. 0.0d+0) then
    dl = dl - ci
    ti = 1.0 / dl
    ql = ql - 0.5d+0*ci*phiir
    phi(i) = ql * ti
    ti = 0.0d+0
    if (tj .ne. 0.0d+0) tj = 2.0d+0*phi(i) - phijb(i,lk,mi)
    if (tk .ne. 0.0d+0) tk = 2.0d+0*phi(i) - phikb(i,j,mi)
    ifixed = 1
  endif
  if (tj .lt. 0.0d+0) then
    dl = dl - cj
    tj = 1.0 / dl
    ql = ql - 0.5d+0*cj*phijb(i,lk,mi)
    phi(i) = ql * tj
    tj = 0.0d+0
    if (tk .ne. 0.) tk = 2.0d+0*phi(i) - phikb(i,j,mi)
    if (ti .ne. 0.) ti = 2.0d+0*phi(i) - phiir
    ifixed = 1
    go to 111
  endif
  if (tk .lt. 0.0d+0) then
    dl = dl - ck
    tk = 1.0 / dl
    ql = ql - 0.5d+0*ck*phikb(i,j,mi)
    phi(i) = ql * tk
    tk = 0.0d+0
    if (ti .ne. 0.0d+0) ti = 2.0d+0*phi(i) - phiir
    if (tj .ne. 0.0d+0) tj = 2.0d+0*phi(i) - phijb(i,lk,mi)
    ifixed = 1
    go to 111
  endif
  phiir          = ti
  phii(i)        = phiir
  phijb(i,lk,mi) = tj
  phikb(i,j,mi)  = tk
  jfixed = jfixed + ifixed
END DO ! i
```

5.11 Parallelization Spectroscopy Summary

In this subsection, we summarize the spectroscopic analysis of the parallelization of the production HPC codes we studied (listed in Table 1). We report the $\mathcal{S}_{cov}(\mathbf{L}, T)$ metric as it is representative of the performance potential of a technique under consideration.

Benchmark	$\mathcal{S}_{cov}(\mathbf{L}, T)$				
	Reduction	Privatization	Loop Transformations	Symbolic Analysis	Call-site Analysis
AMG	✗	1.0	✗	✗	✗
CrystalMk	0.52	0.81	✗	✗	✗
IRSmk	✗	1.0	✗	✗	✗
CPMD	✗	0.91	✗	0.91	0.91
POP	✗	0.46	✗	✗	0.54
UMT2K	✗	0.67	✗	✗	✗
RF-CTH	✗	0.72	✗	✗	✗
SPPM	✗	1.0	✗	✗	✗
HYCOM	✗	0.69	✗	✗	✗
Sweep3d	✗	1.0	✗	✗	✗

Table 5. Summary of parallelization spectroscopy

From Table 5 and the detailed case-by-case analysis presented earlier in this section we make the following conclusions:

❒ Existing techniques for program parallelization are "sufficient" for extracting most of the TLP (from coverage standpoint) available in production HPC codes. The remaining TLP could not be extracted using the existing techniques because of, but not limited to, the presence I/O as in the case of RF-CTH or due to may-dependences as in the case of UMT2K or due to dependences with a very small dependence distance. A relatively small value of $\mathcal{S}_{cov}(\mathbf{L}, T)$ in the case of HYCOM can be attributed to high (15%) coverage of library calls. Although other techniques such as thread-level speculation (TLS) can be employed for parallelizing HPC codes beyond what can be done using the existing techniques, the speedup achievable via such techniques would be small, as evidenced from the high (close to a maximum value of 1.0) of $\mathcal{S}_{cov}(\mathbf{L}, T)$. This is akin to the results reported by Bova et al. for an Euler flow code [8], i.e., most of the TLP in a HPC code can be harnessed in a non-speculative fashion (via OpenMP/MPI directives) and with very little source code alteration.

❒ There is a critical need to develop richer program semantics to guide the compiler in determining which program transformations to apply and for run-time parallelization. This would minimize the sensitivity of program parallelization with respect to the strength of dependence analysis of the particular compiler used. For example, augmenting the existing set of OpenMP directives to support run-time dependence checks can enable parallelization of the hot loop in CPMD.

❒ Better expressivity of the inherent TLP at the programming language level is required to assist the compiler. Recent efforts to this end are exemplified by support for explicit modeling of iteration spaces in the Chapel [10] programming language.

❒ Support for feedback to the user is required, akin to the technique proposed by Wu et al. in [58], to assist algorithmic or application-level transformation for program parallelization.

From Table 5 we note that, for the applications we studied, no loop transformations such as loop peeling, loop permuation were required to *enable* thread-level parallel execution. For instance, loop permutation is not warranted to parallelize the triply nested loop shown in subsection 5.3. Likewise, loop distribution is not required to enable parallelize the multi-way loop [43] shown in subsection 5.9. Of course, this need not be true for all the HPC codes. Further, such loop transformations can potentially assist in achieving higher level of TLP. For example, let us consider a doubly nested `DOALL` loop wherein the outer loop has 4 iterations and the inner loop has 10,000 iterations. Given an octal core machine, the outer loop cannot be parallelized into 8 threads. In order to alleviate this limitation, the loops can be interchanged, which would enable 8-way parallelization of the outer loop in the transformed loop nest. In a similar vein, loop tiling [57] can be employed to exploit temporal and spatial locality (as applicable) thereby improving the overall multithreaded performance.

6. Related Work

Weinberg et al. proposed a methodology to obtain architecture-neutral characterizations of the spatial and temporal locality exhibited by the memory access patterns of HPC applications [56]. Cheveresan et al. presented a comparative analysis of HPC workloads in [11]. In particular, they studied characteristics such as (a) instruction decomposition (floating-point and integer, loads, stores, branches and software prefetch instructions), (b) temporal and spatial locality, (c) sensitivity with respect to cache size cache associativity, (d) data sharing analysis and (e) efficacy of data prefetching. In [38], Nagarajan et al. presented a scheme for proactive fault tolerance for arbitrary MPI codes, wherein processes automatically migrate from "unhealthy" nodes to healthy ones. Their scheme leverages virtualization techniques combined with health monitoring and load-based migration.

In [8], Bova et al. describe their experiences converting an existing serial production code to a parallel code combining both MPI and OpenMP. The scope of the paper is restricted to a harbor response simulation code. Likewise, techniques for hybrid par-

[4] The function calls in the outer loops account for less 2% of the total coverage!

allelization – based on MPI/OpenMP or MPI/Pthreads – of applications ranging from molecular dynamics [20], costal wave analysis [35], atmospheric research [33] and computational fluid dynamics [13] have been proposed. In [18], Gropp et al. described their performance tuning experiences with a 3-d unstructured grid Euler flow code from NASA. In [53], Verma et al. investigate the use of power management techniques for HPC applications on modern power-efficient servers with virtualization support. They showed that for HPC applications, working set size is a key parameter to take care of while placing applications on virtualized servers. None of the aforementioned works address evaluation of the available TLP in HPC applications and parallelization spectroscopy. We believe that our work is complimentary to the above.

Recently, Bridges et. al [9] and Zhong et al. [60] presented techniques for uncovering thread-level parallelism in sequential codes. Akin to our work, Zhong et al. explored the applicability of a set of transformations, such as, variable privatization, reduction variable expansion, speculative loop fission and speculative prematerilization, to parallelization of applications in the SPEC CPU benchmark suite. Contrary to our experimental methodology, their results are simulation-based and further, they assume a perfect memory system. Given that they address non-HPC codes, we believe that their work is complimentary to the work presented in this paper.

7. Conclusion

In this paper, we present a detailed measured analysis of the available thread-level parallelism in production codes. The codes are industrial or are widely used publicly available applications. The measurement was done on two different, viz., POWER5 and Xeon, architectures. Based on the analysis, we draw the following conclusions:

- First, our measurement and analysis shows that more than 75% of the total coverage is inherently parallel in the applications we studied. The corresponding program regions, loops in the current context, can be marked parallel using OpenMP [40] pragmas.
- Second, for applications such as POP, parallelism is available at higher levels, i.e., beyond the lowest level of a calling context. Therefore, higher level program analysis is necessary to assess the *true* coverage of the IP category.
- Third, the benchmarks listed in Table 1 do *not* use parallel packages (wherever applicable) for routines such as Cholesky factorization [1] or FFT [14]. This has two pitfalls: (i) it subjects the parallelization of these routines to the strength of dependence analysis of the specific compiler used and (ii) the code generated by the compiler does not measure up with the code of the hand-tuned packages. Thus, the use of these libraries is imperative from both performance and productivity perspective.
- Fourth, as evident from Section 5, library routines account for a significant percentage of the total coverage in many benchmarks. From this we conclude that parallelization of libraries such as `libc` would assist in achieving better performance in many applications.

As future work, we plan to study the cache performance of the applications during concurrent execution.

References

[1] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Proceedings of the 6th International Euro-Par Conference*, pages 368–378, Aug. 2000.

[2] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: opportunities and limits. In *Proceedings of the 12th ACM International Conference on Supercomputing*, pages 101–108, Melbourne, Australia, 1998.

[3] A. M. Amin, M. Thottethodi, T. N. Vijaykumar, S. Wereley, and S. C. Jacobson. Aquacore: A programmable architecture for microfluidics. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 254–265, San Diego, CA, 2007.

[4] U. Banerjee. *Loop Transformation for Restructuring Compilers*. Kluwer Academic Publishers, 1993.

[5] U. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, 1997.

[6] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–57, New York, NY, 2006.

[7] P. Bose. Workload characterization: A key aspect of microarchitecture design. *IEEE Micro*, 26(2):5–6, 2006.

[8] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, and H. A. Gabb. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *International Journal on High Performance Computing Applications*, 14(1):49–64, 2000.

[9] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–84, 2007.

[10] Chapel. http://chapel.cs.washington.edu/.

[11] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In *Proceedings of the 21st ACM International Conference on Supercomputing*, pages 73–82, 2007.

[12] CPMD Consortium page. http://www.cpmd.org.

[13] S. Dong and G. E. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–17, Baltimore, MD, 2002.

[14] FFTW. http://www.fftw.org/.

[15] Project Fortress Overview. http://research.sun.com/projects/plrg/Fortress/overview.html.

[16] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[17] GNU gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

[18] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. Smith. Performance modeling and tuning of an unstructured mesh CFD application. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 34, 2000.

[19] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[20] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 10, 2000.

[21] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

[22] HPC Programming for the Masses. http://www.hpcwire.com/blogs/17897359.html.

[23] HYCOM Consortium page. http://hycom.rsmas.miami.edu/.

[24] Teraflops Research Chip. http://www.intel.com/research/platform/terascale/teraflops.htm.

[25] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[26] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of the 20th ACM International Conference on Supercomputing*, pages 24–35, Cairns, Australia, 2006.

[27] B. Kreaseck, D. Tullsen, and B. Calder. Limits of task-based parallelism in irregular applications. pages 43–58, 2000.

[28] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the SIGPLAN '07 Conference on Programming Language Design and Implementation*, pages 211–222, San Diego, CA, 2007.

[29] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 162–173, San Diego, CA, 2007.

[30] LAM MPI Parallel Computing. http://www.lam-mpi.org/.

[31] LAPACK – (Linear Algebra PACKage). http://www.netlib.org/lapack/.

[32] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 313–322, Washington, D. C., 1992.

[33] R. D. Loft, S. J. Thomas, and J. M. Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 18–18, Denver, CO, 2001.

[34] S. F. Lundstrom and G. H. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 19–27, Aug. 1980.

[35] P. Luong, C. P. Breshears, and L. N. Ly. Coastal ocean modeling of the u.s. west coast with multiblock grid and dual-level parallelism. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 9–9, Denver, CO, 2001.

[36] H. McGhan. SPEC CPU2006 Benchmark Suite. In *Microprocessor Report*, Oct. 2006.

[37] S. Muchnick. *Advanced Compiler Design Implementation*. Second edition, 2000.

[38] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the 21st ACM International Conference on Supercomputing*, pages 23–32, Seattle, Washington, 2007.

[39] NetCDF (network Common Data Form). http://www.unidata.ucar.edu/software/netcdf/.

[40] OpenMP Specification, version 2.5. http://www.openmp.org/drupal/mp-documents/spec25.pdf.

[41] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, Newport Beach, CA, Oct. 1999.

[42] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 412–423, San Diego, CA, 2007.

[43] C. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 235–242, Aug. 1987.

[44] AMG source code. http://www.llnl.gov/asc/sequoia/benchmarks/amg2006_v0.9.1.tar.gz.

[45] ASC Sequoia Benchmark Codes. http://www.llnl.gov/asc/sequoia/benchmarks/.

[46] Crystalmk source code. http://www.llnl.gov/asc/sequoia/benchmarks/CrystalMk_v0.9.0.tar.

[47] IRSmk source code. http://www.llnl.gov/asc/sequoia/benchmarks/IRSmk_v0.9.0.tar.

[48] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 1–12, San Diego, CA, 2007.

[49] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, Ligure, Italy, 1995.

[50] SPEC CPU Benchmarks. http://www.spec.org/benchmarks.html.

[51] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Aug. 1993.

[52] UPC. http://upc.gwu.edu/.

[53] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of HPC applications. In *Proceedings of the 22th ACM International Conference on Supercomputing*, pages 175–184, Island of Kos, Greece, 2008.

[54] C. von Praun, R. Bordawekar, and C. Caşcaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, Salt Lake City, UT, 2008.

[55] S. Wang, X. Dai, K. S. Yellajyosula, A. Zhai, and P.-C. Yew. Loop selection for thread-level speculation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Aug 2005.

[56] J. Weinberg, M. McCracken, A. Snavely, and E. Strohmeir. Quantifying locality in the memory access patterns of HPC applications. In *Supercomputing*, Nov. 2005.

[57] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Nov. 1989.

[58] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, Alberta, Canada, 2008.

[59] The X10 Programming Language. http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html.

[60] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, February 2008.