

## Parallel Processing Enhancement to SWMM/EXTRAN

Edward H. Burgess, William R. Magro, Michael A. Clement,  
Charles I. Moore, and James T. Smullen

Modifications have been made to the FORTRAN source code of the EXTRAN block of SWMM, which enable the model to take advantage of parallel processors for faster program execution during runtime. These modifications have been made to the program code which performs the explicit (Modified Euler) solution of the St. Venant equations for computation of flow and head within the modeled drainage network. The code changes are designed to support use of OpenMP (see Bibliography) parallel processing directives when the code is compiled using specialized parallel processing compiler extensions (KAP/Pro Toolset for OpenMP). Code changes were verified for correct parallelization and model output confirmed by testing against output produced with the serial (unmodified) version of the same source code. Model output and runtimes were characterized for two relatively large model networks (386 and 772 conduits) by executing the serial and parallelized code on the same hardware (Windows® NT workstation running dual Pentium® 200 MHz microprocessors). Runtime reductions on the order of 30-37% were found for the parallelized code on this commonly available dual processor system. The modified code and OpenMP support an unlimited number of parallel processors, and greater runtime reductions are expected for more highly parallel systems (e.g. those with four or more processors).

---

Burgess, E., W.R. Magro, M. Clement, C. Moore and J. Smullen. 2000. "Parallel Processing Enhancement to SWMM/EXTRAN." *Journal of Water Management Modeling* R206-03. doi: 10.14796/JWMM.R206-03.

© CHI 2000 www.chijournal.org ISSN: 2292-6062 (Formerly in Applied Modeling of Urban Water Systems. ISBN: 0-9683681-3-1)

### 3.1 Introduction

Initial efforts by Camp Dresser & McKee (CDM) to develop runtime reduction modifications to EXTRAN involved runtime profiling performed by J. E. Edinger Associates to quantify the allocation of typical execution time across EXTRAN's various subroutines. This work revealed that approximately 86% of the typical execution time for the EXTRAN executable code is expended in execution of the XROUTE subroutine (or YROUTE, including calls to related subroutines) which perform the numerical solution of the dynamic flow equation to compute the flow and head time series values reported in the EXTRAN output (Buchak, 1996). This runtime profiling work in turn led to the completion of modifications to EXTRAN which enable the solution of the full dynamic flow equation (XROUTE or YROUTE) to be suspended during continuous simulations, when user-defined flow conditions are met (i.e. tolerances used to define when steady-state conditions can be assumed) during the simulation. (Burgess, 1997; also see comment lines and documentation files with EXTRAN source code).

The modifications to suspend XROUTE/YROUTE during steady-state conditions enable significantly faster continuous simulations to be performed. Testing on the combined sewer interceptors in Philadelphia has established that computational speed increases by a factor of four or more can be expected (Burgess, 1997). However, this modification does not affect the runtime of the EXTRAN model during wet weather periods in continuous simulations, nor does it have any effect on runtimes for event-based applications of the model. For these reasons, efforts were undertaken to explore other avenues for further runtime reductions with EXTRAN. These efforts continued to focus on the XROUTE/YROUTE subroutines.

### 3.2 Source Code Modifications, Testing and Tuning

SWMM/EXTRAN employs the Modified Euler method to perform an explicit numerical solution to the dynamic flow equation. The original Modified Euler solution is coded into EXTRAN as the XROUTE subroutine (variable ISOL=0 on the B0 card). Later, a second formulation of the finite difference form of the dynamic flow equation was developed, which applies a different derivation of the convective acceleration term in the momentum equation. This formulation was developed to enable better numerical stability in the solution while allowing for longer time steps to be used. This formulation, known as the Enhanced Explicit Solution, was coded with the same Modified Euler explicit numerical solution into a second subroutine (YROUTE; variable ISOL=1 on the B0 card) that may be used in place of the original explicit solution (XROUTE).

A third formulation of the numerical solution of the dynamic flow equations is included in Version 4 of EXTRAN, known as the Iterative Explicit Solution (ZROUTE; variable ISOL=2 on the B0 card). This formulation uses the same finite difference form of the dynamic flow equation, but modifies it to include weighting coefficients. The weighting coefficients are developed for use in an underrelaxation iterative matrix solution which performs the numerical integration of the flow and head equations. The initial efforts to parallelize EXTRAN have focused on development of parallel processing modifications to the XROUTE subroutine, which were easily ported to the very similar YROUTE subroutine. However, due to the significantly different formulation of ZROUTE, this subroutine has not yet been parallelized.

### 3.2.1 Parallel SWMM/EXTRAN

Parallelism was introduced into the SWMM/EXTRAN model by modifying only the innermost loops. The following summarizes the computation in simple terms. SWMM/EXTRAN performs a series of calculations within of a series of nested loops, where the outer loop is over the individual time steps required to complete the simulations. This looping is controlled within subroutine TRANSX. For each time step, the program calls XROUTE or YROUTE to compute flows in each conduit. These routines compute flow and depth of flow at the end of the time step using the modified Euler explicit solution method.

Even on a machine with multiple processors, a program will use only one CPU at a time unless specifically written to take advantage of multiple processors. The key to successfully utilizing multiple processors is to identify sections of the code or data that can be simultaneously computed. Once this so-called “parallel” work is identified, multiple independent threads can run on distinct processors to divide the work. If the work is truly independent, and the order of floating point operations is not significant, the program results will not change, but the run time will decrease. This is the goal of parallel processing.

Parallel programs, that is those that utilize multiple CPUs, come in many forms, including hand-threaded, message-passing, and directive-based. Although each form consists of multiple “threads” of control working to solve a common problem, the latter is by far the simplest to use. Unlike hand threading and message passing, in which the programmer must explicitly control each thread and the work division, parallel processing directives allow the programmer to describe, rather than implement, the parallelism via program comments, or “directives”. The parallel compiler, then, implements the parallelism described in these directives. The result of this approach is that the parallel directives annotate the serial source code, leaving the code in a state that compiles with parallel and traditional serial compilers alike. Further, the details of thread creation, management, and work division are controlled by the compiler and its run-time library rather than the programmer, leading to dramatically lower parallel development and maintenance costs.

In the SWMM/EXTRAN effort, we used the parallel processing directives specified in the OpenMP standard. OpenMP directives enable cross-platform portability and high performance in parallel applications. The OpenMP model is that of multiple threads executing in a single shared address space, so OpenMP programs are portable across shared-memory multi-processor computers.

OpenMP provides, among other things, directives to create and destroy threads, divide work in loops, synchronize access to shared variables, and create private copies of work variables. We used each of these features in the parallelization of SWMM/EXTRAN. When compiled with an OpenMP-compliant compiler, an OpenMP program can run in parallel. We used the Kuck and Associates' (KAI) Guide compiler, part of the KAP/Pro Toolset, for this purpose.

To locate the best loops to target for parallel speedups, we first profiled the code, confirming that XROUTE, YROUTE, and the subroutines they call consume the majority of the run time. Parallelizing these routines led to the greatest reductions in run time. To understand how the parallelism works, we should first outline the structure of SWMM/EXTRAN. The XROUTE and YROUTE routines perform the following steps:

1. Compute half-step discharge at  $t + \Delta t/2$  in all links based on the preceding full-step values of head at connecting junctions.
2. Compute half-step weir, orifice, and pump flow transfers at  $t + \Delta t/2$  based on preceding full-step heads.
3. Compute half-step head at all nodes at time  $t + \Delta t/2$  based on average of preceding full-step and current half-step discharges in connecting conduits, plus half-step flow transfers.
4. Compute full-step discharge in all links at time  $t + \Delta t$  based on half-step heads.
5. Compute full-step flow weir, orifice, and pump transfers at  $t + \Delta t$  based on half-step heads.
6. Compute full-step head at  $t + \Delta t$  for all nodes based on average of preceding full-step and current full-step discharge, plus flow transfers at current full-step.

Steps 1 and 4 involve loops over all conduits whereas step 3 and 6 is a loop over all junctions. Steps 2 and 5 are performed in subroutine BOUND.

Steps 1 and 4 loop over all conduits, computing flows based on heads computed previously. The loops call subroutine HEAD, which returns the representative conduit cross-sectional area and radius used in the momentum equation. HEAD also determines the portion of the conduit surface area assigned to the upstream and downstream junctions.

The work for each conduit is nearly independent, though a few interdependences exist in the computation of AS, SUMQ, SUMQS, and SUMAL. For each of these, a separate value is kept for each junction, and each conduit updates two of these, one for the upstream junction and one for the downstream junction.

A dependence between iterations exists because multiple conduits can feed into a given junction, leading to the possibility of two threads attempting to simultaneously update a single memory location. This type of dependence is known as a reduction.

To remove the reduction dependence and thereby expose parallelism, private copies of each of these variables were created for each thread. OpenMP provides a simple mechanism to create parallel threads with private storage. The directive:

```
"!$OMP PARALLEL PRIVATE(AS, SUMQ, SUMQS, SUMAL)"
```

creates multiple threads with private copies of AS, SUMQ, SUMQS, and SUMAL, but with shared access to the remaining variables. (OpenMP directives always begin with "\$OMP" or "C\$OMP" to distinguish themselves from ordinary Fortran comments).

The OpenMP PRIVATE ( ) clause is not valid for global variables, and the variable AS was originally in a common block. To allow private copies of AS for each thread, this variable was promoted from a common block variable to a formal argument of the subroutine HEAD. Each thread initializes its private storage and accumulates its contributions into that storage. At the end of the loop, the private copies are combined to produce the final, shared values. To ensure that only one thread modifies the shared copies at any given time, an OpenMP critical section surrounds the lines of code that update the variables.

For simple scalar reductions, OpenMP provides a REDUCTION ( ) clause, which automatically creates the required private variable and combines them after the loop. Figure 3.1 shows a simple serial reduction and its parallel equivalent in OpenMP.

<pre>SUM = 0.0 DO I = 1, N     SUM = SUM + A(I) ENDDO</pre>	<pre>SUM = 0.0 !\$OMP PARALLEL DO REDUCTION(+:SUM) DO I = 1, N     SUM = SUM + A(I) END DO</pre>
---	--

**Figure 3.1** A simple Fortran scalar reduction and its OpenMP parallel equivalent.

The reductions in SWMM/EXTRAN are not of such a simple form, however, since each conduit contributes to the flows and head in two junctions. This form is known as an array reduction, and OpenMP does not have a simple syntax to express such reductions. To parallelize XROUTE and YROUTE, we coded the parallel reductions by hand, as shown in Figure 3.2. First, the shared copy of SUMAL is initialized to zero. Next, parallel threads are created with private copies of SUMAL1, which has the same shape as SUMAL. These private data are then initialized to zero. The threads split the work in the loop following the "\$OMP DO" directive, each accumulating results into its private copy of

<pre> DO J = 1, NJ   SUMAL(J) = 0.0 END DO DO N = 1, NTC   NL = NJUNC(N,1)   NH = NJUNC(N,2)   ...   SUMAL(NL) = SUMAL(NL) + X   SUMAL(NH) = SUMAL(NH) + X ENDDO </pre>	<pre> DO J = 1, NJ   SUMAL(J) = 0.0 END DO !\$OMP PARALLEL PRIVATE(SUMAL1,NH,NL) DO J = 1, NJ   SUMAL1(J) = 0.0 END DO !\$OMP DO DO N = 1, NTC   NL = NJUNC(N,1)   NH = NJUNC(N,2)   ...   SUMAL1(NL) = SUMAL1(NL) + X   SUMAL1(NH) = SUMAL1(NH) + X END DO !\$OMP END DO NOWAIT !\$OMP CRITICAL DO J = 1, NJ   SUMAL(J) = SUMAL(J) + SUMAL1(J) END DO !\$OMP END CRITICAL !\$OMP END PARALLEL </pre>
---	---

**Figure 3.2** An array reduction, like those found in SWMM/EXTRAN, and its OpenMP parallel equivalent.

SUMAL1. Finally, the threads add their partial sums to the shared variable SUMAL. The “!\$OMP CRITICAL” directive ensures that only one thread updates SUMAL at a time.

Step number 3 computes heads and volumes at each node. Because the values in any node do not affect the computation of the values at any other node, the calculation at each node is nearly independent. There is, however, interdependence between the iterations of these loops, due to the calculation of the surcharge tolerance, which accumulates over junctions. Consequently, these steps were not parallelized. The source, however, could be modified to remove the dependence and allow additional parallel speedup.

### 3.2.2 Correctness Testing

Like normal Fortran program statements, OpenMP directives are explicit instructions to the compiler, and they must be specified correctly to ensure a correct parallel program. The most common parallel programming errors include marking a loop as parallel, even though it contains dependences between iterations, and misclassifying private data as shared, or vice-versa.

Traditionally, programmers have ensured the correctness of their parallel programs by manually tracking the use of each local variable, formal argument, and common block variable through each parallel region. This process is tedious and error-prone, since variables must be followed down and back up the call chain, where their names often change as they cross subroutine boundaries. Further, every common-block variable use must be checked, since such variables rarely appear explicitly in argument lists.

A more modern approach is to use an automated debugger. To rapidly verify the correctness of parallel SWMM/EXTRAN, we used a specialized tool from KAI, called Assure. Like an OpenMP compiler, Assure understands the OpenMP directives in the Fortran source code. However, instead of creating a parallel program as its output, Assure creates an ideal parallel computer simulation. The resulting program is then run with a real data set. As it runs, the simulator compares the original serial program with its worst-case behavior on a real parallel computer and systematically locates any OpenMP programming errors.

A companion tool, AssureView, presents the results of the simulation in a graphical user interface. Unlike a traditional debugger, AssureView locates the exact line in the code where the error originated, rather than the location of the symptom. This combination of systematic and detailed debugging information made rapid parallelization of SWMM/EXTRAN possible.

Figure 3.3 shows a typical AssureView session. In the top pane, the highlighted error text, “WRITE → WRITE NL in XROUTE”, indicates an error involving variable NL in subroutine XROUTE. A “WRITE → WRITE” condition means two or more threads could simultaneously attempt to write to the shared variable NL. The lower pane shows that the error occurs at line 63 of the source file XROUTE.FOR. In this case NL is a work variable holding the index of the current conduit’s downstream junction, so the solution is to make NL private to each thread, which is accomplished by adding NL to the PRIVATE list at line 48. Likewise, the variable NH should be private.

Once we had fully parallelized SWMM/EXTRAN, we ran multiple data sets through Assure to ensure that no more parallel programming errors were present. Following completion of the Assure testing, the next level of testing was conducted on the model output to ensure the numerical consistency of results between the parallelized and unmodified (serial) code. This testing is described below in Section 3.3.

### 3.2.3 Performance Tuning

After we debugged parallel SWMM/EXTRAN, we timed the original serial and the new parallel versions of the program. To time the codes, we compared the elapsed wall clock times for the two versions and computed a parallel speedup by taking the ratio of the serial to parallel times. The speedup, however, was less than we had anticipated based on the amount of parallel code, which indicated a parallel performance problem.

Parallel program performance often falls short of expectations for a number of reasons, including: insufficient fraction of time in parallel code; work distributed unevenly across threads; too many small parallel loops; excessive time inside critical sections; and saturated memory bandwidth on the underlying computer hardware.

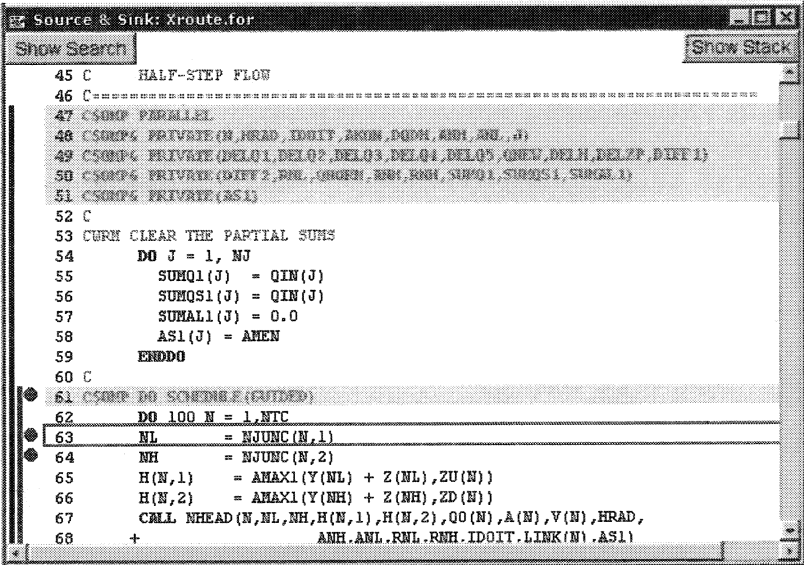
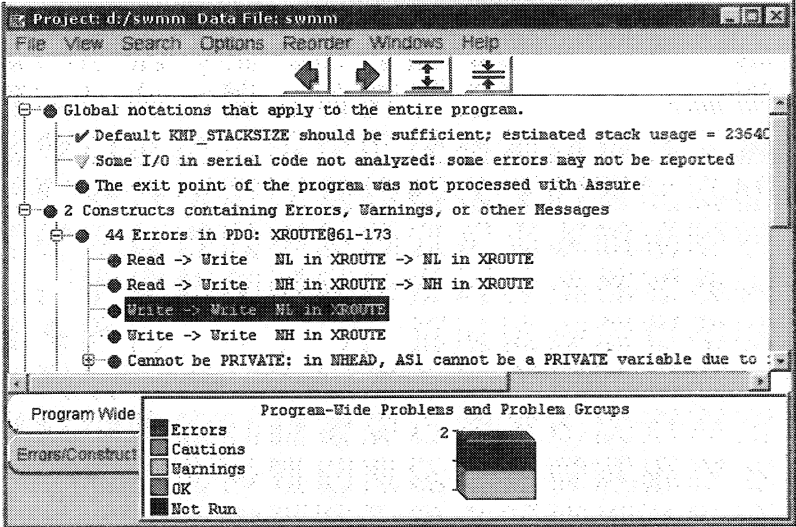


Figure 3.3 A typical AssureView session. AssureView displays the location of parallel programming errors, located automatically by Assure.

To discover the root of the performance problem, we used another KAI tool, called GuideView. GuideView is a specialized OpenMP parallel performance tool. Unlike traditional program profilers, which measure CPU time, GuideView measures elapsed wall clock times of parallel program sections. It also tracks the

relative timing of parallel program events, which is critical to program performance. For example, a parallel program can run twice as fast on two threads if and only if the work is split evenly and the independent tasks occur simultaneously. If one of the tasks is delayed, then the overall performance will suffer. GuideView explicitly measures such delays and identifies the cause.

The GuideView system consists of two components: an instrumented parallel run-time library and a graphical user interface. To use the special library, we simply relinked the SWMM/EXTRAN executable using the Guide compiler. Next, we ran SWMM/EXTRAN with real data sets on one and then two processors. In each case, the instrumented library created a program statistics report upon program completion. Finally, we viewed the performance reports side-by-side in the GuideView graphical user interface.

GuideView quickly demonstrated that the two large parallel loops in each of XROUTE and YROUTE were suffering from load imbalance. Load imbalance occurs when the iterations of a loop take different amounts of time to compute, so a simple half-and-half division of the work leads to an uneven distribution of work across threads. Fortunately, OpenMP provides a simple mechanism to correct such load imbalances. To change the way iterations are assigned to threads, one adds a “`SCHEDULE()`” clause to the parallel loop directive. OpenMP provides a number of scheduling options, in addition to its default even-distribution rule. Dynamic scheduling, one such option, initially hands out small chunks of iterations to each thread. When a thread finishes the assigned work, it returns for another chunk. This pattern is continued until all work has been assigned and completed.

We obtained the best parallel performance in SWMM/EXTRAN by using a variant of dynamic scheduling, known as “guided” scheduling. In guided scheduling, each thread initially obtains a large chunk of iterations to complete. As the threads return for new work assignments, each receives a chunk of iterations exponentially smaller than the last, until a minimum size is reached. Guided scheduling’s advantage is that the work is divided into fewer chunks, relative to pure dynamic scheduling, which lowers the fraction of time spent in the parallel library and improves parallel performance.

Normally, changing the mapping of iterations to threads in a parallel program would involve extensive recoding. This type of performance tuning, however, is quite simple with OpenMP. To specify guided scheduling, for example, we simply added “`SCHEDULE(GUIDED)`” to the parallel loop directives. The OpenMP compiler, Guide, and its run-time library automatically made the appropriate changes to the parallel program to implement our loop scheduling choice. After the change, the loops over conduits in XROUTE and YROUTE were nearly perfectly balanced across threads.

To further increase the parallel speedups in SWMM/EXTRAN, additional portions of code must be parallelized.

### 3.2.4 Numerical Considerations

In principle, one can perform perfectly independent tasks simultaneously or in an arbitrary order and still obtain identical results. A digital computer, however, is not ideal, and one area where it approximates reality is its representation of real numbers. The IEEE standard for floating-point numbers stipulates that a real number should be represented by the closest number exactly representable by a fixed number of data bits. SWMM/EXTRAN uses double precision arithmetic, so its floating point numbers are represented by 64 bits.

The result of this inexact representation, is that some mathematical properties of real numbers no longer hold in a computer. For example, the associative property of addition stipulates that:

$$\exp(1) + \exp(2) + \exp(3) + \exp(4) + \exp(5) + \exp(6) = \\ [\exp(1) + \exp(2) + \exp(3)] + [\exp(4) + \exp(5) + \exp(6)].$$

On a digital computer, however, the left and right sides will differ by a tiny, but measurable amount. The right hand side, however, is exactly how a parallel program with two threads would compute this sum. Although the differences are small, over the course of a long run, these small differences can accumulate and lead to macroscopic differences in the final program results.

Naturally, neither the left-hand side, as computed by a serial program, or the right-hand side, as computed by a parallel program, is inherently more accurate. Thus, if the final answers obtained by a serial and parallel program with identical input differ in a substantive way, then the underlying model itself and its numerical implementation must be called into question.

The parallel loops in SWMM/EXTRAN contain reductions as discussed above, and the parallel computation of these exploits the associative property of addition. Thus, parallel SWMM/EXTRAN can and often will obtain slightly different answers from the serial version. We performed extensive testing to confirm that the differences were not substantive. This testing is described below in the next section.

## 3.3 Verification Process and Results

EXTRAN models developed as part of the Philadelphia Water Department (PWD) CSO Abatement Strategy were utilized in the initial effort to characterize the numerical consistency and decreases in runtime associated with the SWMMOMP executable. Simulations were performed using the existing SWMM executable, SWMM44DV, and the dual processor parallel executable, SWMMOMP.EXE. Comparisons of model runtimes and model output were

performed for the two executables. Model simulations were performed using a Micron Millennia Pro 200 MHz with dual processors (Windows NT Workstation).

Two modeled systems were utilized to document the effectiveness of the SWMMOMP executable. Table 3.1 lists the key model elements included in each system.

**Table 3.1** EXTRAN elements included in modeled systems.

Modeled system	Number of EXTRAN elements in each modeled system					
	Conduits	Junctions	Storage junctions	Pumps	Outfall junctions	Orifices (1)
High Level	386	375	45	-	17	-
Low Level	772	758	47	1	39	20, 26

Note:

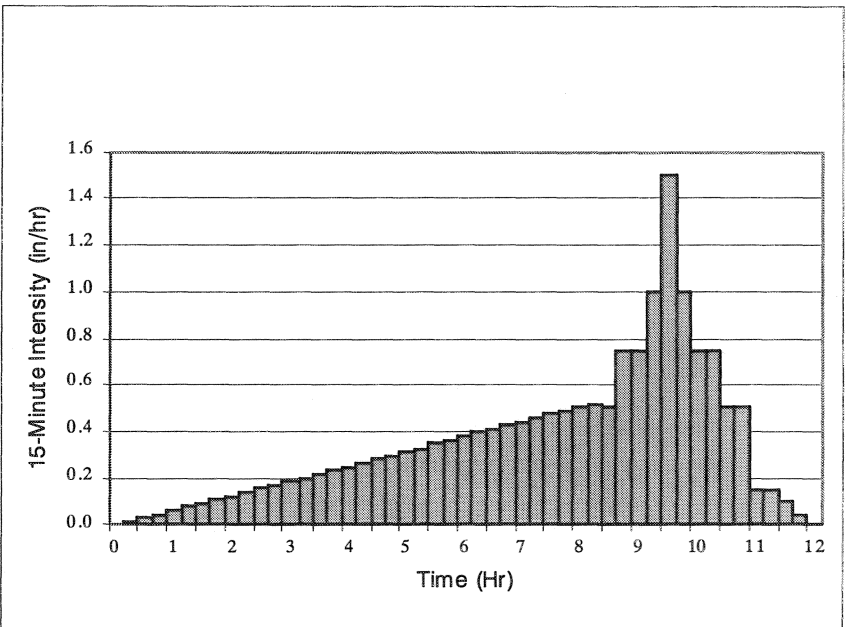
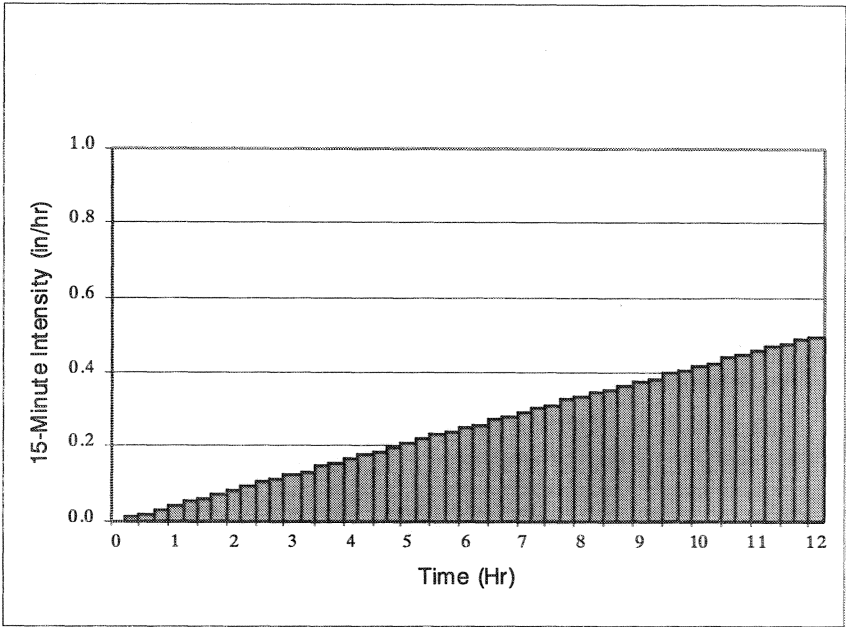
(1) Low Level System includes head dependant variable orifices, standard sump orifices.

Diversion weirs were modeled as hydraulically equivalent (over the flow range of interest) conduits developed external to EXTRAN and input into both modeled systems. Free outfalls were applied as model boundary conditions to outfalls in the High Level System. A time-varying tidal series was applied as a model boundary condition to outfalls in the Low Level System.

Two artificial wet weather events were created to test the effectiveness of the SWMMOMP executable. Hydrographs were developed using the SWMM RUNOFF block. Hyetograph #1 is a straight ramp hyetograph from 0-0.50 inches/hour over a 12-hour period. Hyetograph #2 is a synthetic event hyetograph with straight ramp intensity for the first 8 hours, a maximum intensity of 1.5 inches/hour at hour 9.5 and a decreasing intensity from hours 10-12. These artificial wet weather events are plotted on Figures 3.4a and 3.4b.

Twelve-hour EXTRAN simulations were performed for the two wet weather events for the High Level and Low Level systems using the SWMMOMP and SWMM44DV executables. Numerical results and runtimes of the SWMMOMP executable were compared against the unmodified code. Heads and flows were compared throughout the modeled network. A statistical analysis of differences in model outputs was performed at select junctions and conduits representative of the various hydraulic structures in the modeled system to quantify the effects of the modified code on numerical results for each analyzed junction and conduit. In order to reduce the output datasets to a manageable size, initial analysis was performed on subsets created with a 15 time step (75 seconds) increment, reducing each output set from 8640 to 576 individual values for both flow and head, for both wet weather events for each of the two executables.

Tables 3.2a and 3.2b list the mean and selected percentile values (1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99%) in the ranked distribution of differences between SWMM44DV and SWMMOMP for the two wet weather



Figures 3.4a and 3.4b Wet weather events #1 and #2.

**Table 3.2a** Model Output Comparison: Simulated Depth with SWMMDV vs.SWMMOMP

Junction	Mean	Differences in distribution of modeled depths (ft.) at specified percentiles								
		99%	95%	90%	75%	50%	25%	10%	5%	1%
31900	0.00	0.05	0.01	0.01	0.00	0.00	0.00	-0.01	-0.01	-0.02
34110	0.00	0.07	0.02	0.01	0.00	0.00	0.00	-0.01	-0.01	-0.03
D45	0.00	0.25	0.02	0.00	0.00	0.00	0.00	0.00	-0.03	-0.31
44010	0.01	0.33	0.04	0.02	0.00	0.00	0.00	-0.01	-0.04	-0.17
41040	-0.03	0.29	0.06	0.03	0.00	0.00	-0.01	-0.11	-0.17	-0.39

**Table 3.2b** Model Output Comparison: Simulated Flow Rate with SWMMDV vs.SWMMOMP

Conduit	Mean	Differences in distribution of modeled flows (cfs) at specified percentiles								
		99%	95%	90%	75%	50%	25%	10%	5%	1%
31900	0.02	0.80	0.18	0.08	0.03	0.00	-0.02	-0.06	-0.11	-0.23
34115	0.02	0.52	0.17	0.10	0.03	0.00	-0.02	-0.06	-0.10	-0.28
ORF#20	0.00	0.21	0.15	0.12	0.00	0.00	0.00	-0.11	-0.14	-0.20
40675	0.00	0.21	0.01	0.01	0.00	0.00	0.00	0.00	-0.01	-0.17
42608	0.00	0.81	0.28	0.10	0.01	0.00	-0.01	-0.06	-0.23	-1.25
ORF#26	0.00	0.11	0.04	0.02	0.00	0.00	0.00	-0.02	-0.05	-0.13

events for the selected junctions and conduits. Figures 3.5a and 3.5b display the distribution of differences in model output for the selected junctions and conduits.

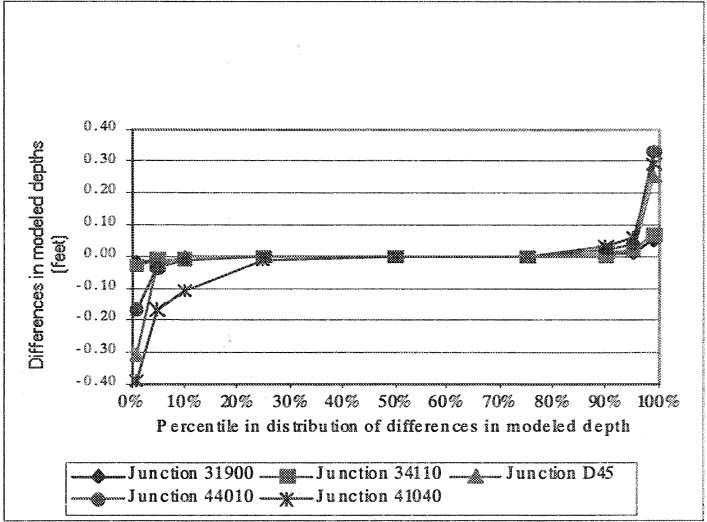


Figure 3.5a Distribution of differences in model estimated depth at selected junctions for two wet weather events using two different executables.

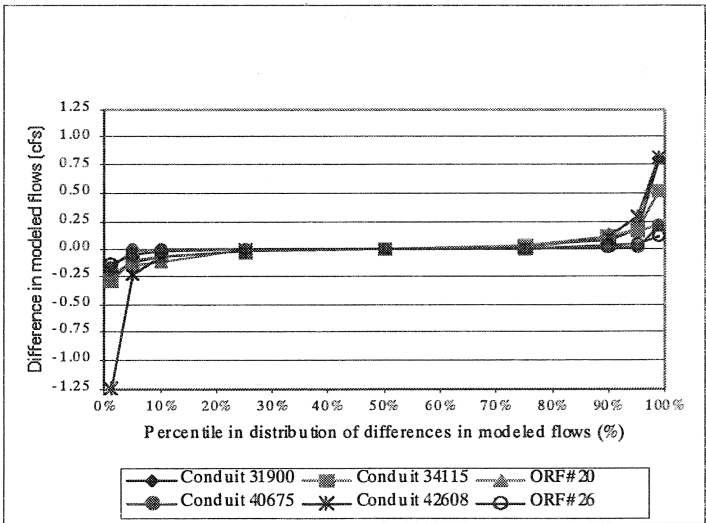


Figure 3.5b Distribution of differences in model estimated flow at select conduits for two wet weather events using two executables.

The model output comparisons were tabulated by computing the difference in depth (Table 3.2a) and flow rate (Table 3.2b) at each incremental time step for the two output files (one computed with SWMMOMP and the other with SWMM44DV) at selected junctions/conduits. This produced a time series of \*h and \*q, which were each re-ordered to produce a ranked list (from highest, or largest positive difference, to lowest, or largest negative difference) for each junction/conduit. The ranked differences at each specified level (1% through 99%) were both tabulated (Tables 3.2 a/b) and plotted graphically (Figures 3.5a/b).

The results presented in Tables 3.2a/b and Figures 3.5a/b indicate generally insignificant differences in modeled heads and flows using the two executables. However, the differences in modeled heads and flows were found to occasionally be significant in some simulations. Further examination revealed that these non-negligible differences were generally limited to the brief periods of mild numerical instability typical of many complex closed-conduit model networks (e.g. as a “dry” pipe just begins to fill) or “solution instability” (e.g. as various portions of the system reach the surcharge threshold, when brief periods of surcharge-free surface solution switching can occur). The modified code enables parallel processing by performing operations in a different sequence than the unmodified serial code. Therefore, different sequencing of operations can allow for head and flow values to be output by the two codes with non-negligible differences when the model solution is unstable. Fortunately, this condition is minimized or eliminated in properly developed system models and therefore these differences in output can be considered inconsequential.

A comparison of model continuity error was performed to further quantify effects of the modified code on the results by evaluating output differences from the two codes on a system-wide basis. Table 3.3 compares model continuity error for the two wet weather events using the modified and unmodified code. Table 3.3 indicates generally insignificant differences in the two executables on a system-wide basis. The only difference that was considered non-negligible was that for Event #2 in the High Level system, where slightly greater model instability was observed than for the other event/system.

**Table 3.3** Comparison of model continuity errors using the SWMMOMP and SWMM44DV executables.

Collection System	Wet weather event	SWMM44DV Continuity error (%)	SWMMOMP Continuity error (%)
High Level	1	-4.49	-4.4
High Level	2	-2.67	-2.16
Low Level	1	-0.82	-0.84
Low Level	2	-0.47	-0.48

### 3.4 Performance Testing Results

Model run times are compared for the two wet weather events using the two executables in Table 3.4, which indicates significant decreases in simulation time (30-37%) for both modeled systems due to the parallel executable.

**Table 3.4** Comparison of Model Run Time Using the SWMMOMP and SWMM44DV Executables.

Collection System	Wet weather event	SWMM44DV	SWMMOMP	Percent decrease in run time (%)
		Run time (minutes)	Run time (minutes)	
High Level	1	7	4.4	37%
High Level	2	7.9	5.3	33%
Low Level	1	13.8	9.7	30%
Low Level	2	18.7	13.1	30%

### 3.5 Conclusion

This analysis of the SWMM44DV and SWMMOMP executables using two complex EXTRAN models resulted in significant ( $\geq 30\%$ ) decreases in runtime using the SWMMOMP executable without sacrificing numerical resolution. Insignificant differences exist between the model output generated with the two codes (although significant differences in modeled flow rates and depths have been found to occur during periods of model instability). Additional testing with simulations of other model networks and other flow conditions are planned to provide further confirmation of model reliability and speed enhancements. At this point, only XROUTE and YROUTE have been parallelized, although other areas of the code could be modified to produce further runtime reductions and further investigation of these opportunities is planned.

### Bibliography

- Buchak, E.M.; personal communication; January 29, 1996.
- Burgess, E.H.; internal project memorandum; June 19, 1997; Long-Term CSO Control Plan Project- Philadelphia CSO Program.
- Kuck & Associates; Assure Version 3.6 Reference Manual; February 1999.
- Kuck & Associates; Guide Version 3.6 Reference Manual; February 1999.
- OpenMP FORTRAN Application Program Interface, Version 1.0; OpenMP Architecture Review Board (<http://www.openmp.org>); October 1997.
- Roesner, L.A., J.A. Aldrich, and R.E. Dickinson; Storm Water Management Model User's Manual Version 4- Addendum I: EXTRAN; Aug 1988; US EPA, Athens, GA.